COMPILER GUIDED SCHEDULING : A CROSS-STACK APPROACH FOR PERFORMANCE ELICITATION

A Dissertation
Presented to
The Academic Faculty

By

Girish Mururu

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

December 2020

COMPILER GUIDED SCHEDULING : A CROSS-STACK APPROACH FOR PERFORMANCE ELICITATION

Approved by:

Dr. Santosh Pande, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Ada Gavrilovska
School of Computer Science
Georgia Institute of Technology

Dr. Umakishore Ramachandran
School of Computer Science
Georgia Institute of Technology

Dr. Vivek Sarkar
School of Computer Science
Georgia Institute of Technology

Dr. Tushar Krishna
School of ECE
Georgia Institute of Technology

Date Approved: August 10, 2020

Winners are not those who never fail but those who never quit.

APJ Abdul Kalam

To my parents, Nagappa Naik and Indira Naik

ACKNOWLEDGEMENTS

To begin with, I would like to thank my advisor Dr. Santosh Pande for providing me with the opportunity to work on several interesting problems. Not only the problems in this thesis, but he also introduced me to several different domains such as security and reliability. I have enjoyed exploring these new domains and have learnt a lot. I believe he has instilled in me the confidence to research in different areas for which I am very grateful. His support and advice on all the problems that I worked on and also when faced with rejections from conferences has been invaluable. I would also like to thank Dr. Ada Gavrilovska, who always spared time to discuss the system problems that I worked on. I am thankful to my other committee members– Dr. Umakishore Ramachandran, Dr. Vivek Sarkar, and Dr. Tushar Krishna, for their valuable feedback on my thesis.

It was always a learning experience filled with fun when working with my lab-mates. I would like to thank Chris, Chao, Sharjeel, Kaushik, Prithayan, Vincent, Amit, and Vinit for working with me. I hope you all enjoyed my company as much as I did yours. I enjoyed every moment I spent at Georgia Tech. The faculty at Georgia Tech create an atmosphere of learning and I consider myself very lucky for getting an opportunity to attend various classes and learn a lot of fascinating things. I also had access to the amazing infrastructure in terms of research equipment, buildings, and facilities at Georgia Tech. I am thankful to the badminton club at Georgia Tech, a fun-filled sports club, where I made some great friends.

I am very lucky to have many friends and they have been supportive of my endeavors. I am thankful for the funny, light-hearted conversations we sporadically had that always cheered me up during my Ph.D. I would like to thank Brahma, Mahesha, Macchi, Deekshit, Ramanan, Deeksha, Nitesh, Kp, Chotu, Patil, Guru, Manoja, and Vaibhav.

Finally, yet importantly, I would like to thank my family for their unconditional love and support. I thank my wife-to-be Sneha, who has been nothing but supportive while pursuing

my Ph.D. I am thankful to my brother Harsha, whose journey as an entrepreneur has inspired me, for all the tech conversations and my sister-in-law Soumya, who is a very good friend of mine. I am also thankful for the priceless moments with my nephew Atharva (Dindima) and cheerful conversations with my cousins Sumeet and Sourabh during my Ph.D. I have always been inspired by my father Nagappa Naik from his knowledge and experience and my mother Indira Naik from her passion for teaching. They have stood by me and supported my decisions creating an environment for pursuing my dreams. I dedicate this thesis to my family.

TABLE OF CONTENTS

Acknowledgments v

List of Tables xi

List of Figures xii

Summary xvi

Chapter 1: Introduction 1

1.1 Compiler Guided Scheduling 2

1.1.1 Thesis Statement 3

1.1.2 Contributions 3

1.1.3 Problem of Process Migration 7

1.1.4 Interface between the Compiler and Scheduler 8

1.1.5 Problem of Process Co-Scheduling 9

1.1.6 Problem of Process Co-Location 11

1.1.7 Problem of Secure Co-Execution 12

Chapter 2: Overview on Completely Fair Scheduler (CFS) 14

Chapter 3: PinIt: Influencing OS Scheduling via Compiler-Induced Affinities . . 17

3.1 Relevant Background 18

3.2 PinIt Framework 19
3.3 PinIt Analysis 21
3.3.1 Memory Reuse Density 21
3.4 PinIt Optimization 25
3.4.1 Call Hoisting 25
3.4.2 Loop Transformation 28
3.5 Pin Threshold 30
3.6 Pinit Runtime 31
3.7 Experiments 32
3.7.1 Experimental setup 32
3.7.2 Experimental goals 34
3.7.3 Experimental Results 35
3.8 Related Work 41
3.9 Conclusion 43

Chapter 4: Beacons: A Compiler - Scheduler Interface for Dynamic Attributes . 44

4.1 Relevant Background 45
4.2 Beacons Framework 45
4.3 Loop Timing Model Generation 47
4.4 Memory Footprint Analysis 50
4.4.1 Calculating Memory Footprint of a Loop 50
4.4.2 Classifying Reuse and Streaming Loops 52
4.5 Beacons 53

4.5.1 Beacon Classification 53
4.5.2 Beacon Hoisting 55
4.5.3 Beacon Insertion 55
4.6 Evaluation 56
4.6.1 Benchmarks 57
4.6.2 Timing Accuracy 58
4.7 Related Work 60
4.8 Conclusion 60

Chapter 5: Maximizing Throughput through Beacon based Co-Scheduling . . . 61
5.1 Relevant Background 62
5.2 Beacon Enabled Scheduler (BES) 62
5.3 Experiments 67
5.4 Related Work 71
5.5 Conclusions 72

Chapter 6: Minimizing Latency with Fairness through Beacons 73
6.1 Relevant Background 74
6.2 Bellator 74
6.3 Experiments 93
6.4 Related Work 108
6.5 Conclusion 109

Chapter 7: Secure co-execution using Beacons: thwarting side-channel attacks . 110

7.1 Problem Motivation and Solution 110
7.1.1 Cache-based Side-channel Attack Techniques 111
7.1.2 Defense Mechanisms 113
7.1.3 Beacon-based Solution 117
7.2 Conclusion 118

Chapter 8: Conclusion 120
8.1 Usage of the system 123
8.2 Future Work 124
8.3 Other Works 125

References 135

LIST OF TABLES

3.1 Reuse Density Threshold 30

3.2 Shared pin library characteristics 32

3.3 Configuration of experiment machines 33

3.4 Mixes of high-priority applications in MediaBench 33

3.5 Application characteristics 34

3.6 Average batch throughput for 16threads(8threads) normalized to priority CFS 35

3.7 Average latency for 16threads(8threads) normalized to priority CFS 35

3.8 Djpeg before and after loop splitting optimization 40

4.1 Overhead of Beacon Library Calls 56

4.2 Configuration of machines used for experiments 57

LIST OF FIGURES

1.1 Compiler Guided Scheduling 6

2.1 A machine with four sockets with each socket containing 10 cores. The four dotted rectangles denotes the scheduling domains with respect to the first core in the machine. 15

3.1 Steps to minimal pinning in PinIt. 21

3.2 hoisting outside external loops 28

3.3 Mediabench: speedup of high-priority applications normalized to priority CFS vs PinIt 35

3.4 Mediabench: cache Misses (MPKI) in priority CFS vs PinIt 36

3.5 Mediabench: page faults in priority CFS vs PinIt 36

3.6 Mediabench: migrations in priority CFS vs PinIt 37

3.7 sd-vbs: speedup of high-priority applications normalized to priority CFS vs PinIt 37

3.8 sd-vbs: page faults in priority CFS vs PinIt 38

3.9 sd-vbs: migrations in priority CFS vs PinIt 38

3.10 sd-vbs: stalls in priority CFS vs PinIt 38

3.11 Average normalized speedup: Pinit extracts the best of scheduler flexibility and processor affinity 40

4.1 Beacon Compilation Component 46

4.2 Timing Accuracy of differnet types 58

4.3 Average Timing Accuracy 58

5.1 Beacon Enabled Scheduler (BES) listens to beacons to co-schedule processes . . . 63

5.2 Different timing Scenarios of Incoming Beacon 64

5.3 A simplified Mealy state machine of the beacon scheduler. Key: **B**eacon, **R**euse, **S**treaming, **F**iller, **J**ob, **C**omplete, **T**hreshold, **deq**ueue, **enq**ueue, **$**(cache) 65

5.4 Throughput Normalized to CFS on ThunderX 68

5.5 Throughput Normalized to CFS on ThunderX2 69

5.6 Memory Accesses Normalized to CFS on ThunderX2 69

5.7 Histograms for the job completion times of CFS and the beacon-enabled scheduler (BES) for Cholesky (left) and correlation (right). The X-axis represents discrete timesteps, and the Y-axis is a count of the number of jobs that completed within a given timestep. 69

5.8 Throughput of Beacon Scheduler and Reactive Baseline normalized to CFS on ThunderX2 70

6.1 Bellator listens to beacons and effectively co-locates processes 75

6.2 Speedup (100 th percentile) for 27 processes job configuration 93

6.3 Speedup at 25, 50, 75 th percentile of processes completion for 27 processes job configuration 94

6.4 Cache Behavior in terms of cache misses, L3 reads and hits for 27 processes job configuration 94

6.5 Fairness in load distribution among L3 caches for 27 processes job configuration 95

6.6 Speedup (100 th percentile) for 54 processes job configuration 95

6.7 Speedup at 25, 50, 75 th percentile of processes completion for 54 processes job configuration 96

6.8 Cache Behavior in terms of cache misses, L3 reads and hits for 54 processes job configuration 96

6.9 Fairness in load distribution among L3 caches for 54 processes job configuration 97

6.10 Speedup (100 th percentile) for 108 processes job configuration 97

6.11 Speedup at 25, 50, 75 th percentile of processes completion for 108 processes job configuration 98

6.12 Cache Behavior in terms of cache misses, L3 reads and hits for 108 processes job configuration 98

6.13 Fairness in load distribution among L3 caches for 108 processes job configuration 99

6.14 Speedup (100 th percentile) for 162 processes job configuration 99

6.15 Speedup at 25, 50, 75 th percentile of processes completion for 162 processes job configuration 100

6.16 Cache Behavior in terms of cache misses, L3 reads and hits for 162 processes job configuration 100

6.17 Fairness in load distribution among L3 caches for 162 processes job configuration 101

6.18 Speedup (100 th percentile) for 216 processes job configuration 101

6.19 Speedup at 25, 50, 75 th percentile of processes completion for 216 processes job configuration 102

6.20 Cache Behavior in terms of cache misses, L3 reads and hits for 216 processes job configuration 102

6.21 Fairness in load distribution among L3 caches for 216 processes job configuration 103

6.22 Speedup (100 th percentile) for 50 with xl processes job configuration 103

6.23 Speedup at 25, 50, 75 th percentile of process completion for 50 with xl processes job configuration 104

6.24 Cache Behavior in terms of cache misses, L3 reads and hits for 50 with xl processes job configuration 104

6.25 Fairness in load distribution among L3 caches for 50 with xl processes job configuration 105

7.1 Log Normalized Cache Misses of Attacks 117

SUMMARY

Modern software executes on multi-core systems that share resources like several levels of memory hierarchy (caches, main memory, secondary storage), I/O devices, and network interfaces. In such a co-execution environment, the performance of modern software is critically affected because of resource conflicts arising from sharing of these resources. The resource requirements vary not only across the processes but also during the execution of a given process. Current resource management techniques involving OS schedulers have evolved from and mainly rely on the principles of fairness (achieved through time-multiplexing) and load-balancing and are oblivious to the dynamic resource requirements of individual processes. On the other hand, compiler research has traditionally evolved around optimizing single and multi-threaded programs limited to one process. However, compilers can analyze the process resource requirements. This thesis contends that a significant performance enhancement can be achieved through the compiler guidance of schedulers in terms of dynamic program characteristics and resource needs.

Towards compiler guided scheduling, we first look at the problem of process migration. For load-balancing purposes, OS schedulers such as CFS can migrate threads when they are in the middle of an intense memory reuse region thus destroying warmed up caches, TLBs. To solve this problem while providing enough flexibility for load-balancing, we propose PinIt, which first determines the regions of a program in which the process should be pinned onto a core so that adverse migrations causing excessive cache and TLB misses are avoided. The thesis proposes new measures such as unique memory reuse and memory reuse density, that capture the performance penalties incurred due to migration. The compiler analysis determines program regions to be pinned and pin/unpin calls are then hoisted at the entry and exits of the region; the migrations being prevented in pinned regions. In an overloaded environment, compared to priority-cfs, PinIt speeds up high-priority applications in mediabench workloads by 1.16x and 2.12x and in computer vision-based workloads by

1.35x and 1.23x on 8 cores and 16 cores, respectively, with almost same or better throughput for low-priority applications.

The problem of co-scheduling and co-location of processes that share resources must be solved for efficiency in a co-execution environment. Towards this, several approaches proposed in the literature rely on static profile data or dynamic performance counter based information, which inherently cannot be used in an anticipatory (proactive) manner leading to suboptimal scheduling. This thesis proposes Beacons, a generic framework that instruments the programs with generated models or equations of specific characteristics of the program and provides a runtime counterpart that delivers the dynamically generated information to the scheduler. We develop a novel timing analysis for the duration of the loop that is on average 84% accurate on Polybench and Rodinia benchmarks and embed that along with memory footprint, and locality classification information into beacons. The thesis presents two schedulers, one that targets the problem of co-scheduling maximizing throughput called Beacon Enabled Scheduler(BES), and the other that targets the problem of co-location minimizing latency with fairness called Bellator. A prototype of BES improves throughput over the default Linux scheduler (CFS) by up to 4.7x on ThunderX and up to 5.2x on ThunderX2 servers for consolidated workloads. A prototype of Bellator on ThunderX2 with 224 hardware threads achieves lower 100^{th} percentile latency by 14% on average while executing 108 and 162 simultaneous processes and by 3% on average for 54 and 216 simultaneous processes.

The thesis provides a preview of how beacons with cache misses information, modeled similar to the timing analysis, can enable secure co-location of processes in a multi-tenant environment by detecting and mitigating cache-based side-channel attacks. Our beacon-based scheduler solution detects and mitigates attacks through all well-known cache-based side-channel techniques – Prime+Probe, Flush+Reload, Flush+Flush– on OpenSSL cryptography algorithms in multi-tenant environments.

CHAPTER 1
INTRODUCTION

Modern multi/many-core systems are typically equipped with shared resources such as several levels of memory hierarchy– caches, main memory, persistent storage–, I/O devices, and network interfaces. These shared resources critically dictate the performance of the software applications that concurrently execute on these multi-, many-core systems. The operating system (OS) scheduler tries to efficiently schedule multiple processes and manage shared resources. Different computing environments such as user desktops, servers, depend on these schedulers to meet their computing goals while utilizing these systems efficiently.

State-of-the-art schedulers such as Completely-Fair Scheduler (CFS), the default scheduler in Linux, are completely aloof to individual process requirements and treat every process similarly. However, not only the the resource obligations are different across different processes but also varies within an individual process's execution cycle. The resource requirements or behavior of the processes are unknown to the scheduler. Compilers, however, can analyze these program characteristics, but traditionally compiler optimizations have focused on analyzing and optimizing the performance of the individual (single or multi-threaded) applications with tremendous success in this regard. On the other hand, schedulers have dealt with the problem of resource sharing by mostly adopting fairness as the primary criterion in terms of time-multiplexing in single-core sharing while augmenting the same with load balancing and processor affinity in multi-core scheduling. Both the compiler and the scheduler stacks have continued to evolve stand-alone in the above manner; this thesis claims that there is a significant opportunity to improve performance (throughput and latency) and share resources more efficiently and securely by developing a synergistic approach that bolsters the scheduler with compiler-generated dynamic application attributes to undertake smart scheduling decisions.

1.1 Compiler Guided Scheduling

Modern workloads exhibit highly variant resource usage that depends on phases within the application. All the attributes of resource demands, such as resource type (e.g. cache), duration (how long a particular resource will be under demand by the application), and sensitivity (how sensitive the application is to a given resource such as cache), can vary during the application's execution and different phases can exhibit different characteristics. While some applications exhibit periodicity and repetition with regard to the application phases and the corresponding attributes, many others are highly input data-dependent and thus exhibit no periodicity or regularity in the application phase behavior.

Currently, state of the art research techniques [1, 2, 3] rely on resource usage history or current resource contention (using hardware performance counters) and then react to contention by invoking suitable scheduling mechanisms to correct it, however such reactive mechanisms can be harmful than beneficial. Such approaches have the following major limitations and do not work for many workloads due to the following reasons:

• Many modern workloads are input data-dependent and do not exhibit highly regular, repetitive behavior. Thus for any kind of non-repetitive workload, history-based methods are not accurate predictors.

• In general, reactive approaches take time to detect a resource-heavy phase and this leads to at least two related drawbacks. First, by the time the phase is detected, it may be too late to act (because the phase may be ending). Second, by the time detection takes place, the application state has already bloated (for example its cache consumption). This can impact other processes' cache states, and it can also make it prohibitively expensive to do anything about the offender (because of the cache affinity that can be lost by migrating or pausing).

• Detection-based approaches cannot predict the duration of the phase, nor the sensitivity of its forthcoming resource usage. We show in our work that such a prediction is key to enable proactive decision-making.

These limitations can be overcome by providing the process's behavioral information, generated by the compiler, to the scheduler. Compiler's knowledge about applications have been previously used in runtime mostly tailored to specific runtimes and goals. For example, compiler-generated information has been used in cluster level task management [4], Big-Little heterogeneous architecture process management [5], energy-aware scheduling [6]. However, to the best of our knowledge no prior work has used a compiler to aid general scheduling by generating dynamic attributes such as duration of resource usage or any other resource usage information through which scheduler can attain insights about processes' behavior and infer interference with demands of other co-executing processes to perform intelligent scheduling. Such a lack of communication of dynamic information between an executing application and the OS scheduler has left a significant gap between the two. This thesis proposes to bridge the gap by developing a cross-stack approach using the existing system's interfaces without introducing new ones resulting in a layering solution without modifying the OS, a design envisioned not to perturb other systems' properties that have evolved over a period of time.

1.1.1 Thesis Statement

"Predictive compiler analysis can provide effective dynamic information regarding applications' behaviors and resource requirements to the OS scheduler to enable proactive decision making in terms of efficient and secure resource provisioning leading to significant improvements in performance and security."

1.1.2 Contributions

Towards compiler guided scheduling, we solve the following problems:

1. **Problem of Process Migration.** OS schedulers like CFS migrate the processes during execution for load balancing among the available cores. When a process is migrated, it loses the memory values brought into the private caches of its previous core. The

loss of performance due to non-cached values can be significant especially if the process was executing in a region of large cache footprint and very high reuse of cached values. This thesis provides a compiler-based solution to avoid migrations in such intense reuse regions of the program yet providing enough flexibility to the scheduler for load-balancing purposes.

2. **Interface between the Compiler and Scheduler.** We develop a generic framework called **Beacons** for passing the dynamic attributes of the program generated with the help of a compiler to the scheduler. Beacons are capable of generating the following types of information: cache footprints, cache reuse, expected execution times of a loop, and the completion signal for a loop. Beacons use dynamic program attributes such as the loop bound and execute a statically generated formula generated by the compiler hoisted at the loop entrance to compute the respective information and relay it to the scheduler.

3. **Problem of Process Co-Scheduling.** Using the beacons framework, we develop a scheduler that deals with what processes must be co-scheduled together in a throughput oriented setting to maximize the throughput of the system. In other words, we use beacons to help solve the problem: given a large number of jobs to be executed on a machine with resource constraints, how do we decide what processes must be co-scheduled together and when such that the throughput of the machine is maximized?

4. **Problem of Process Co-Location.** We look at the problem of how a given set of processes must be placed dynamically to minimize the latency of all the process with fairness. OS schedulers like CFS does not know what order of dynamic-process placement yields efficient resource usage and hence it completely depends on the notion of fairness in terms of time-multiplexing and load balancing. While being fair, we leverage the beacons framework to develop a scheduler that determines better co-location for efficient usage of resources thus minimizing latency. For example,

co-location of two cache-sensitive and cache-footprint heavy processes which will displace each others' data is avoided by the scheduler whereas co-location of a cache heavy and cache-sensitive process and another which is either: cache insensitive (streaming-data) process or cache-light (low-cache footprint) process is preferred.

5. **Problem of Secure Co-execution** When executing in a multi-tenant environment with shared resources, a process's private data can be learnt by snooping on the process's activity through side-channels. We look at how cache-based side-channel attacks can be thwarted in multi-tenant environments leveraging the information from beacons. This thesis just peeks into a solution and its initial results showing the possibility of a complete solution and the versatility of beacons in solving a wide range of problems. The full solution to this problem is beyond the scope of the current thesis.

Our compiler guided scheduling framework is as shown in Figure 1.1. The application source which is converted to LLVM intermediate representation (IR) is fed into our LLVM compiler infrastructure (a set of LLVM passes), which first performs required program analysis consisting of static and profile-data-based analysis and instruments the program to generate process attributes during runtime followed by code optimization to reduce the overhead due to either the number of instrumented calls or from the effect of the instrumented calls. The binary generated now consists of calls that generate dynamic attributes about the process's resource requirements during execution. These are calls to the library, shown as the runtime library in the figure, to which the binary is linked. The library sends the process's information either to the user level scheduler (as in the case of beacon-based solutions), which then calls the system's API to schedule the processes as required or can also directly call the system's API. The operating system such as Linux provides a number of APIs (application programming interface) to influence the process schedule or even modify the schedule of processes as required. This enables our cross-stack approach without modifying the operating system.

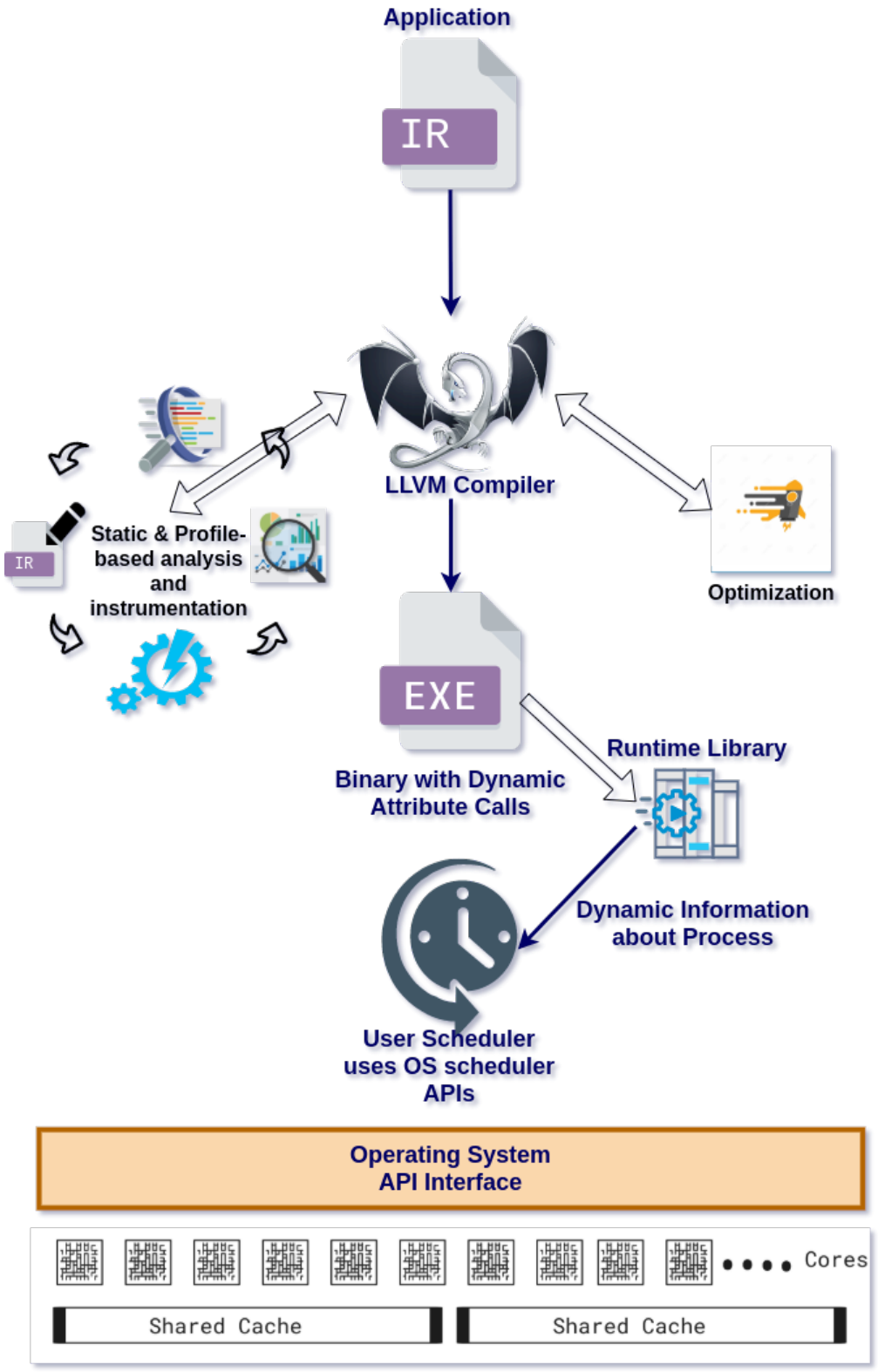


Figure 1.1: Compiler Guided Scheduling

1.1.3 Problem of Process Migration

Modern workloads related to computer vision, media computation and machine-learning exhibit a very high amount of data locality. Although modern OS deploys processor affinity to induce data locality aware scheduling, the lack of precise knowledge of dynamic application characteristics leave a significant performance inefficiency on the table due to a large number of process migrations carried out by the scheduler for load balancing purposes. OS schedulers such as CFS dynamically determine both the CPU time and/or the core that should be allocated to each workload component by time- and space- multiplexing workloads on available cores. As a result, workload threads are *continuously dispatched* on CPU cores , *preempted* after a period of time, and potentially *migrated* onto another core on the same or a different socket on modern multi-core platforms. However, such actions could lead to the loss of cached working set of process, resulting in cache misses, TLB misses, memory stalls, and thus degrading process performance. This effect intensifies when processes are migrated by the OS onto a processor on another socket, which has its own low-level cache (LLC). After all, multiple levels of caches have to be warmed up by transferring data from a socket to a socket over the system interconnect. Therefore, the OS entails a tradeoff between moving the process around for better use of computing resources and improving the cache efficiency by providing processes with undisturbed cache usage, a very common scenario, especially when the number of applications executing exceeds the available resources.

To maximize cache usage, applications have traditionally been optimized in isolation by a compiler [7, 8, 9, 10]. However, because of shared resources and the trade-off mentioned above, modern applications must be optimized as an ensemble executing together; unfortunately, performing global compiler analysis across applications is infeasible. This thesis proposes a novel methodology that limits analysis to individual applications yet induces the operating system to mediate actions across applications that minimize harmful migrations and maintain data locality in the cache, only where and when necessary. The

problem of cache misses is well known to schedulers in OSes such as Linux, which strive to maintain the natural affinity of processes in the execution queue. By scheduling a process on the processor on which it had executed before, the scheduler attempts to maintain cache affinity unless for load-balancing purposes, when the process is scheduled onto a nearby free processor [11]. Such a migration to the nearby free processor will not affect the performance of a process executing in a region with very few access to memory. However, if that region is memory intensive, the effect can be quite adverse [12]. To avoid the effects of migration, hardware resources such as memory can be partitioned such that certain workload components are always executed on a dedicated CPU by pinning (processor affinity). However, pinning the workload components permanently to a given core in an overloaded environment adversely affects the aggregate performance of the workload. To address these challenges, we develop PinIt, a compiler-based technique to determine the regions in the program that exhibit enough reuse and any migration during the execution of these regions degrade the process performance. These regions are then dynamically pinned to the processor. In an overloaded environment, in which the number of processes is greater than the number of core, compared to priority-CFS, PinIt speeds up high-priority applications in mediabench workloads by 1.16x and 2.12x and in vision-based workloads by 1.35x and 1.23x on average on 8cores and 16cores, respectively, while completing the low-priority batch 23% faster on 16 cores in mediabench and almost within the same time for other workloads.

1.1.4 Interface between the Compiler and Scheduler

Next, in this thesis, we develop a more generic beacons framework that can be used by the compiler to instrument the program regions with program characteristics which execute when the program runs and generate dynamic attributes about the code regions. These dynamic attributes are then passed on to the scheduler through a library. We develop novel models for different program characteristics which are useful for scheduling purposes. These

models (characteristics) are not fixed and can be changed as per the problem that is being solved. For performance scheduling, we develop a loop timing model, memory footprint model, and classify the loop as either reuse or stream loops, whereas, for secure scheduling, we replace these models with a cache miss model as we will see later. Our loop timing model is based on machine learning the profiled loop timings and has an accuracy of 84% on average across Polybench and Rodinia benchmark suite. The memory footprint model based on polyhedral methodology statically determines the amount of memory accessed in the loop as a function of loop bounds and the loop is classified as reuse or stream by determining the static reuse distance. The precision of the information carried by beacons varies as per the analyzability of the loops. The scheduler aggregates the beacons of multiple processes and uses the information proactively to respond to the incoming workload and thus performs intelligent scheduling as per the required computing domains. The thesis shows how using the beacons framework, a scheduler can be developed to target different computing domains and properties starting with a performance scheduler for throughput computing, followed by a generic scheduler that achieves better group latency and fairness than Completely Fair Scheduler, and finally a peek into scheduling for secure execution avoiding cache-based side-channel attacks in a multi-tenant environment.

1.1.5 Problem of Process Co-Scheduling

Using the beacons framework, first, we develop a scheduler that dynamically chooses the process to co-schedule to achieve maximum throughput in a throughput oriented setting with a large number of jobs. Typically, schedulers conservatively co-schedule processes to avoid cache conflicts since miss penalties are quite heavy leading to lower resource utilization ((ranging from 50 to 70%). In a throughput oriented setting, such a conservative scheduling leads to significant losses in terms of achieved throughput. The paradigm of throughput computing, (as outlined in [13]), emphasizes the overall work performed over a fixed period, as opposed to how fast a single core or thread executes a process. Throughput

computing entails high job counts with potentially large data sets. Memory footprints can quickly overwhelm a throughput-oriented server, stressing all levels of its memory hierarchy. Throughput oriented systems continuously perform work and thus must be built with energy-efficient processors. In throughput paradigm, several works [14, 15] have also suggested using processors built for the mobile space for maximizing performance per watt. Currently, the service industries are interested in ARM-based servers for such capabilities. Even on these processors, maximizing throughput by carefully managing the resource contention is a very challenging task, especially for modern data-oriented workloads that are memory bound and extremely sensitive to memory latency.

Fortunately, the working sets of such workloads usually fit in modern caches, thanks to advances in hardware in terms of the larger cache sizes, smarter cache partitioning strategies, as well as advanced compilation techniques that perform loop transformations to take advantage of the data reuse. However, core counts are continuing to grow, and a large number of cores are beginning to emerge that share a cache at the L2 and even L3 level. A great example of such an architecture is the ARM-based ThunderX and its successor, the ThunderX2. They have 48 and 224 hardware threads that share a 16 MB L2 and 32 MB L3 cache, respectively. When many independent processes that do not share any data are scheduled on such a large number of cores that share L2 or L3 cache, the underlying contention can be significant due to large data footprints. This causes the total working set of co-scheduled processes to spill across the last level cache into the main memory, which would lead to a significant degradation in the throughput of scheduled applications. At the heart of the problem is the scheduling-algorithm that makes decisions about what independent processes to co-schedule across the available cores.

The need for better management of these resources for throughput computing is addressed by load-balancing mechanisms employed in parallel programming [16, 17] and at the cluster level in distributed scheduling [18, 19, 20]. However, throughput-oriented, system-level schedulers for many-core machines are rare because of the difficulties in

effective OS-level load balancing [21, 22, 23]. While this problem could be solved to a reasonable degree for stable workloads by a-priori (offline) classification [24]and online management that leverages this classification [2], it does not work satisfactorily for modern workloads that are very much data dependent (e.g. machine learning or data analytic workloads). Our throughput oriented scheduler relies on the dynamic beacon information and also augments imprecise information through the use of performance counters to throttle higher concurrency when needed. The result of such a high throughput oriented scheduler is that it effectively schedules a large number of jobs and improves throughput over CFS by up to 4.7x on ThunderX and up to 5.2x on ThunderX2.

1.1.6 Problem of Process Co-Location

Next, we develop Bellator, a beacon-enabled latency scheduler which tries to minimize resource conflicts among all the executing processes and achieve overall lower latency for the executing processes with fairness. CFS, the Completely Fair Scheduler of Linux, performs fair scheduling by dividing the computing resources among the executing processes. This also achieves overall lower latency, because with fairness equal progression of every process is ensured. While the processes are co-located or migrated for load-balancing and ensuring all processes are executing in fair environments, the lack of knowledge of the process' resource usage characteristics can result in co-locating processes that conflict instead of complement resource usage. Works such as bubble-up [24], autopin [25] have tried to find the best possible co-location offline by analyzing the profile data. These co-locations are fixed and do not cater to either the dynamic needs of the processes or to the dynamic environments in which processes execute. Many works such as Merlin [3] use performance counters and suffer from the drawbacks of reactive approaches mentioned before. However, Merlin deals with the problem of co-locating processes at cluster level and does not deal with scheduling within a machine competing against CFS. Bubble-flux [2] leverages the memory pressure curve developed in bubble-up to co-locate a set of high-priority processes and then

uses low-priority processes to utilize wasted CPU cycles to improve machine utilization but does not compete with CFS in general scheduling through dynamic co-location. Even if CFS were to be augmented with performance counter information, they would still suffer from the delay in detecting phase changes, the lack of knowledge about the amount of resource requirements, and noises. To overcome these limitations, Bellator depends on the beacon information to acquire the knowledge of process' forthcoming resource requirements and co-locates the processes accordingly to minimize the resource usage conflicts. As a result on ThunderX2 while using 224 hardware threads we achieve lower 100^{th} percentile latency by 14% on average while executing 108 and 162 simultaneous processes and by 3% on average for 54 and 216 simultaneous processes.

1.1.7 Problem of Secure Co-Execution

Finally, we show how beacons can be utilized by the scheduler in detecting and mitigating cache-based side-channel attacks that result because of the shared cache resources. Side-channel attacks steal secret keys cleverly leveraging information leakage of various kinds and can therefore break encryption. Detection and mitigation of side-channel attacks is a very important problem. Although this is an active area of research, the solutions proposed in the literature have limitations in that they do not work in a real-world multi-tenancy setting on servers, have high false positives, or have high overheads, thus limiting their applicability. In this thesis, we introduce the idea of a compiler guided scheduler that can leverage beacons to detect with high accuracy cache-based side-channel attacks for processes on multi-tenancy servers. In this work, the beacon generates the information regarding cache-misses pertaining to the code region (forthcoming loop). Beacons convey the cache miss information at runtime to the scheduler which uses it to schedule processes such that their combined cache footprint does not exceed the maximum capacity of the last level cache. The scheduled processes are then monitored for actual vs predicted cache misses, and when an anomaly is detected, the scheduler performs a cache-misses counter-based

search to isolate the attacker. Our beacon-based scheduler can detect and mitigate all attacks from well-known cache-based side-channel attack techniques–Prime+Probe, Flush+Reload, Flush+Flush– on OpenSSL's implementation of cryptography algorithms with no false positives in a multi-tenant environment.

CHAPTER 2

OVERVIEW ON COMPLETELY FAIR SCHEDULER (CFS)

Before we delve deep into the details of the contributions of the thesis, it is helpful to know the working of the default Linux scheduler –The Completely Fair Scheduler (CFS). CFS is most prevalent today in systems ranging from servers to embedded systems and is a major baseline for system scheduling algorithms.

Completely Fair Scheduling first conceptualized in staircase deadline scheduler by Con Kolivas was introduced as Completely Fair Scheduler (CFS) in Linux version 2.6.23. The completely fair scheduler treats a computing resource, a CPU as an entity that can be divided among the different tasks (processes, or thread-group) equally in a fair manner. Although CPU is integral and cannot be physically divided, the computing resource is time-sliced–the time that can be spent executing on a CPU is divided, among the different tasks. CFS maintains a *vruntime* for every process (task) that indicates the amount of time a process has spent on a CPU. Each unit computing resource like a CPU or hardware thread has a runqueue of processes that are sorted by their vruntime and the process with the lowest runtime is the next to be scheduled. CFS has a notion of *target latency*, which is the fixed interval of time that all the processes in the runqueue must execute before another scheduling round. When an executing process is pre-empted to schedule another process a context-switch takes place. Context-switches incur overhead and repeated context-switches can burden the CPU without any actual progress of the tasks in the queue. Hence, CFS runs the processes for at least a time-slice called the *minimum granularity*. For example, on a CPU's runqueue with four processes and a target latency of 20ms, each process is executed for 4ms. However, if the minimum granularity is 5ms then each process is executed that long. CFS also divides the target latency among the processes based on the weighted priority which is defined as niceness. The nice value of each process determines the time-slice of the target latency that

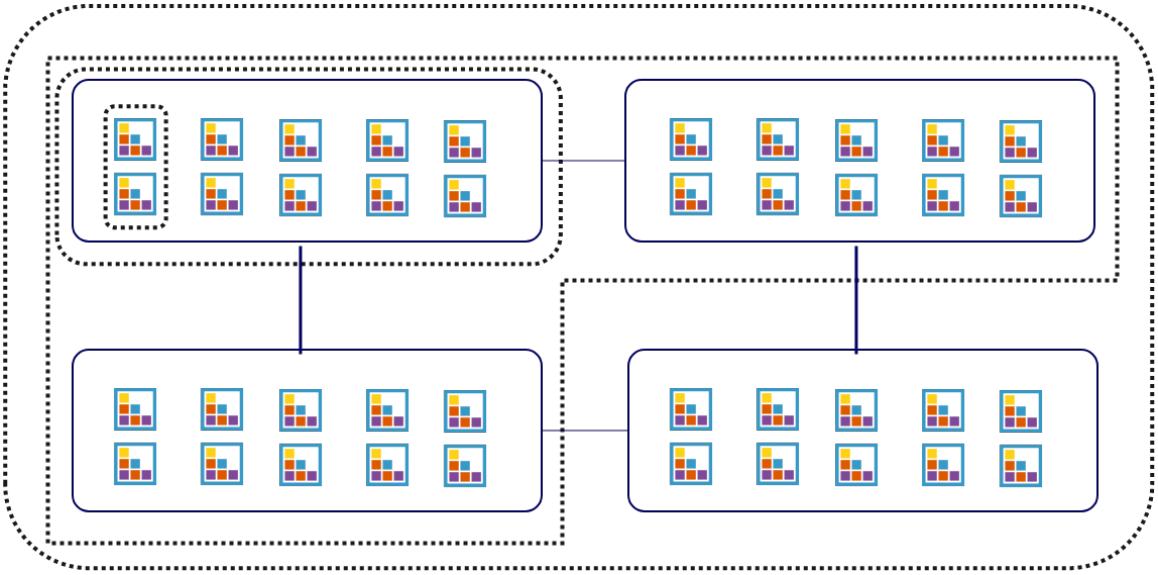


Figure 2.1: A machine with four sockets with each socket containing 10 cores. The four dotted rectangles denotes the scheduling domains with respect to the first core in the machine.

a process receives. The runqueue is implemented as a red-black tree indexed by the process' vruntime.

So far the scheduling mentioned above is what CFS does to multitask on a single CPU. It gets complex with multi-core. Each core or hardware thread (Linux recognizes each hardware thread as a logical CPU) in the case of SMT maintains a runqueue instead of one global runqueue. This avoids synchronization of the queue among multiple CPUs, however, it can lead to imbalanced runqueues. The load-balancing algorithm of CFS shown in Algorithm 1 (as in [26]) is run by each CPU regularly to distribute the load evenly among the different runqueues. For fast load-balancing, the CPUs are organized hierarchically via scheduling domains with the single CPU in the lower-most hierarchy domain as shown in Figure 2.1 [26]. The CPUs next to each other or a core form the next hierarchy, followed by the nearby core pairs, and then the cores in the socket. The sockets with one-hop interconnect in the NUMA configurations form the next domain in the hierarchy, followed by two hops and so on as shown in the Figure with dotted rectangles. In each scheduling domain starting from lowest but one, the load-balancing is performed among scheduling groups formed by the next lower domain. For example, the socket domain balances the load among the

Algorithm 1 Simplified Load Balancing Algorithm in CFS

```
1:  for all sd in sched_domains of cur_cpu do
2:      if sd has idle cores then
3:          first_cpu ← 1ST idle cpu of sd
4:      else
5:          first_cpu ← 1ST cpu of sd
6:      if cur_cpu ≠ first_cpu then
7:          continue
8:      for all sched_group sg in sd do
9:          sg.load ← average loads of CPUs in sg
10:     busiest ← overloaded sg with the highest load
                  (or, if non-existent) imbalanced sg with highest load
                  (or, if non-existent) sg with highest load
11:     local ← sg containing cur_cpu
12:     if busiest.load ≤ local.load then
13:         continue
14:     busiest_cpu ← pick busiest cpu of sg
15:     try to balance load between busiest_cpu and cur_cpu
16:     if load cannot be balanced due to tasksets then
17:         exclude busiest cpu, goto line 14
```

core-pair of the socket. In each scheduling domain of current CPU, an idle CPU, if available, or the first CPU of the scheduling domain is designated to carry out load-balancing (Line 2-5). It first gets the load of the scheduling group containing the current CPU and checks if the load of the busiest scheduling group is less than the current load (Line 8-12). If true, then the load is already balanced at the level, else the load is balanced between the busiest group and the current scheduling group. Load balancing is expensive and CFS frequently balances load whenever there is an idle CPU, if not then it avoids frequent load-balancing unless on process exit or when a new process is spawned in which case it does emergency load-balancing. The load is not just the task weight but also combines the average CPU utilization to avoid idling of computing resources and unfair distribution. Furthermore, threads belonging to one process is treated as a group to avoid unfair advantages to processes with high thread count.

CHAPTER 3

PINIT: INFLUENCING OS SCHEDULING VIA COMPILER-INDUCED AFFINITIES

We first approach the problem of process migration faced in load balancing schedulers on multi-cores. For example, the state of the art default Linux scheduler, CFS, employs process affinity while deciding the CPU on which to execute a process. If a process has previously executed on a certain processor, then CFS schedules the process to execute on the same processor provided the processor is free and the load balancing criteria is satisfied. However, if a processor has more processes than its neighbour, the load balancing algorithm migrates some of the processes to the neighbour to evenly distribute the load. The process that is migrated can be in the middle of a reuse-heavy loop execution and the migration causes the process to lose its working data in the local caches thus hurting the process performance. However, not migrating the process can lead to imbalance and also cost to the performance of the processes in the system. In this chapter, we present PinIt – a compiler-assisted technique, which

1. automatically identifies critical program regions with high cache utilization,
2. instruments the program around critical regions to generate *pin* requests to the underlying scheduler, and then
3. uses this information to dynamically inform the underlying OS scheduler of potentially harmful thread migrations.

The outcome is a framework that migrates processes for resource management only when the system performance is not hurt due to migrations. In developing PinIt we make the following technical contributions:

- We introduce a new metric Memory Reuse Density (MRD) – metric that characterizes cache affinity/property of a program with scheduler flexibility
- Compiler transformation/optimization to insert/hoist the PinIt calls minimizing overhead – the number of calls and avoiding pinning of non-critical regions
- PinIt framework implementation and evaluation that demonstrates practical and effective workload consolidation.

The utility of PinIt is particularly pronounced for those workloads that exhibit heavy memory reuse and locality plays a dominant role, prominent examples of which are media and vision-based applications. Media servers can house media applications serving thousands of users. Users stream media content online and expect low latency. The stored media must be decoded and hence decoders are on this critical path. While the media content is either uploaded by users or media enablers such as Netflix, the media can be encoded offline, and hence media encoders are of lower priority. Similarly, videos and images from many sources can be streamed to a server for analysis by computer vision applications with the expectation of low latency and high throughput from these applications. We evaluated mediabench and san-diego vision-based benchmarks (sd-vbs). For effective work consolidation, we classify the jobs into high -and low -priority applications as in [2]. We run these set of high- and low-priority processes such that the total number of processes is more than the number of processors, which is necessary to exercise scheduling scenarios effectively. Compared to low-priority processes, which is limited to a set of encoders, high-priority processes consist of a set of decoders and vision workloads that are in the critical path. The goal of PinIt is to speedup the high-priority processes while not degrading the throughput of the low-priority processes.

3.1 Relevant Background

There are very few works that use processor affinity let alone induce processor affinity dynamically in the process to prohibit its migration to avoid cache-misses. Autopin [25], an

offline tool similar to Valgrind finds the best thread-to-core mapping through an offline iterative process."Optimal task assignment in multi-threaded processors" [27] also determines the best static assignment of a fixed offline application set. The authors use extreme value theory for generating the best assignment of jobs. These works find the best offline assignment of process or thread to the right core in a fixed setting. These works pin the process during its entire life-cycle to the processor and in a dynamic setting, this hurts the aggregate performance of the system. [28] attempts to assign VCPU for entire execution time to a CPU based on the history of execution and does not dynamically set or reset the affinity of the VCPUs. This is the closet work that addresses the problem of migration, however in the case of VCPUs and not processes in a virtual execution environment. PinIt targets the problem of process migration while executing cache-sensitive regions by influencing the scheduler via compiler induced processor affinity.

3.2 PinIt Framework

The Pinit framework is best described by the goals of the framework that can be summarized as follows:

1. To identify the execution of regions with large cache reuse when scheduling actions can have the most adverse effects.

2. To determine the placement (hoisting) of pin/unpin directives to the underlying scheduler such that intervals within which the flexibility of the scheduler is compromised is minimal and the frequency of the directives (calls to the scheduler) is as low as possible.

Towards these goals, the framework must strive to achieve three objectives.

Maximize reuse density. Concerning the first goal, the sections of code that exhibit large data reuse are loops that can access the same line of memory multiple times within a short span of execution, so the migration of a process when executing such loops is very

harmful. Hence, for pinning to be most influential PinIt must find such loops that have high memory reuse. However, such loops can entail statements that are not transitively closed in underlying dependencies with the statements of high memory reuse. For instance, although belonging to the same loop nest, some statements do not establish any kind of dependency with statements that exhibit memory reuse. Including such independent statements in the execution of pinned loops decreases the average reuse per instruction and also increases the interval within which the scheduler is restricted. Therefore, all memory reuse instructions along with the dependent statements must be grouped into one loop nest and the rest into another nest by splitting the loop. In other words, *one key goal of PinIt is to identify and maximize the density of reuse – i.e., the amount of reuse per unit loop region. PinIt achieves this by finding loops with high reuse per instruction and splitting the loops if possible by removing statements without memory reuse*.

Minimize runtime overheads. To preclude the scheduler from migrating a process that is executing a loop region with high migration cost, our second objective is to request the OS scheduler to pin the process. Since pin/unpin calls to the OS incur overhead, they must be carefully hoisted. The pinning calls should be moved outside the external loops encompassing the pinned regions within. Besides, if multiple pinned regions are neighbors, these regions can be merged thus removing unnecessary intervening pin/unpin calls.

Maintain scheduler flexibility. To maintain the flexibility of the scheduler, PinIt pins the process only to a free processor. The pinning call must first check if the processor on which the process will be pinned is not already reserved by another process. Otherwise, if two processes are pinned to one processor, then one of the processes has to wait while the other is executing, even when the other processors are idling without work. Restricting only one process to a processor ensures that no other processor is starved when processes are waiting to run. The overall flow of the above phases is pictured in Figure 3.1. PinIt Analysis achieves the first goal of finding the pin regions. PinIt optimization splits loops to increase reuse density and optimally places the pin request calls. PinIt library maintains the

flexibility by monitoring and arbitrating the process to the right core.

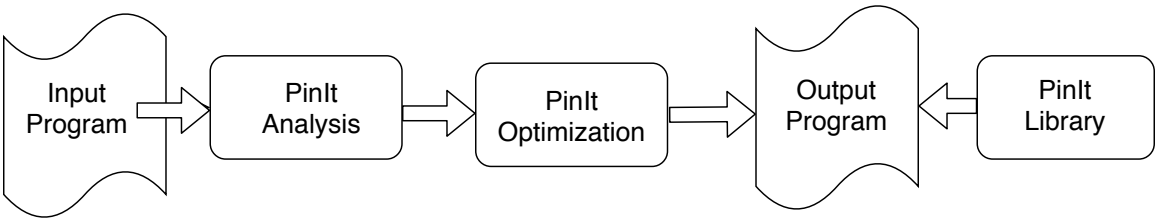


Figure 3.1: Steps to minimal pinning in PinIt.

3.3 PinIt Analysis

We define a few terms to express new properties that serve as a metric for identifying critical loops that should be pinned. Using the metric, we describe the optimization required for pinning applications.

3.3.1 Memory Reuse Density

To determine regions with high memory reuse, PinIt analyzes the dependency between the memory accesses of instructions. Every access to the memory that was accessed earlier results in reuse. The distance or number of loop iterations after which the memory was reused is the reuse distance which is also equal to the dependence distance. We calculate memory reuse by including the reuse distance, because if the reuse distance between memory accesses is higher, then the penalty resulting from migration during such accesses is greater as the number of cache misses is equal to the reuse-distance of the accesses. For example, if a process is migrated while executing a statement, $a[i] = a[i-n]$, subsequent access to $a[i-n]$ from iterations i (value during migration) to $i+n$ results in cache misses. Hence, by considering the reuse distance, memory reuse (MR) is calculated as follows:

$$MR = \Sigma_{i=0}^{n} rd_i, \tag{3.1}$$

where

$$rd_i = \begin{cases} reuse\_distance, & \text{if } reuse\_distance > 0 \\ 1, & \text{otherwise} \end{cases} \tag{3.2}$$

where $reuse\_distance$ are with respect to all other instructions in the loop and Memory Reuse (MR) is the summation of these values over all memory instructions in the loop.

The total memory reuse does not capture the cache penalties due to migration. In other words, loops can have many memory accesses that result in total higher reuse but very little individual reuse which can also be classified as streaming loops. Pinning such streaming loops is not of much value because these hardly require any memory to be solicited on process migration. This contrast is clearly explained below in the Fibonacci and summation examples.

```
1 for(int i=0; i < N; ++i)
2  a[i] = a[i-1] + a[i-2];
```

Code 3.1: Fibonacci

```
1  for(int j = 0; j < M; ++j)
2   for(int i=0; i < j; ++i)
3     b[j] = b[j] + b[i];
```

Code 3.2: Summation

In Fibonacci, $a[i]$, $a[i-1]$, and $a[i-2]$ access each element only three times. If the process is migrated while executing this loop, two array elements must be transferred between the caches for access $a[i-2]$ and one element for access $a[i-1]$, which hardly incurs any cost. However, in summation, $b[i]$ results in $M/2$ accesses of each array element, and every iteration touches $M/2$ array elements on average. If this loop is migrated, then in the worst case, M cache elements must be transferred. To account for the penalty during migration, we define "Unit Memory Reuse (UMR)," which is the memory reuse generated by each

unique access. i.e.

$$UMR = \frac{Memory\_Reuse(MR)}{Unique\_accesses}. \tag{3.3}$$

Unit memory reuse (UMR) is $4 * N/N = 4$ for Fibonacci and $M * N/2 * N = M/2$ for summation thus clearly capturing the migration penalty.

In the absence of any optimization and for large reuse distances where the register allocation cannot be done across array elements, the code produced for Fibonacci sequence generator in Code 3.1 will do the following:

1. Prefetch a in cache
2. Load $a[i-1]$ into a register
3. Load $a[i-2]$ into another register
4. Add the two
5. Store the result into $a[i]$

Consider the above instruction sequence, in the worst case, the context switch and process migration can occur at any or all of the above points. Worst case cache misses when it occurs at (1), (2), (3) or (4), and (5) causing cache misses is as follows:

- At (0) cache was not warm, array a elements were prefetched into the cache causing the first miss.
- At (1) $a[i-1]$'s line was in the cache before migration, now upon migration at (1) the load on $a[i-1]$ causes a miss
- At (2) $a[i-2]$'s line was in the cache before migration, - now the load on $a[i-2]$ causes a cache miss
- At (3) or (4) line $a[i]$ was in the cache before migration - now store on $a[i]$ will cause a cache miss for the write-through cache.

This sequence of migrations can cause a total of four cache misses. So the total reuse accounts worst-case misses due to migrations and hence results in $4 * N$ total misses for the loop and UMR capture the four misses for each access.

The ratio Unit Memory Reuse (UMR) to the total number of instructions in the loop is defined as ***Memory Reuse Density (MRD)*** (Equation 3.4), a measure of the trade-off between reuse and the size of the regions that participate in the reuse. Compared to regions with lower reuse densities, regions that exhibit high MRD are expected to have greater benefit from maintaining their cache content, and should be pinned.

$$MRD = \frac{UMR}{instructions} \tag{3.4}$$

The following code excerpt will help illustrate the above ideas.

```
1 void foo(int M, int N)
2 {
3     ...
4   for(int j = 0; j < M; j++)
5   for(int i = j; i < (j+N); i++)
6   {
7       A[i] = A[i-1] + A[i-2];
8       B[i] = B[i-3] + B[i-2];
9       C[i] = C[i-5] + k;
10      x = i + 2 + x;
11      y = n + i + y;
12      z = z+1;
13      m = m--;
14      k = i + k;
15  }
16  ...
```

17 }

Code 3.3: Function foo()

Within the inner loop, the first statement has three memory accesses in $A[i], A[i-1], A[i-2]$. The write to $A[i]$ is read in the next iteration as $A[i-1]$ and the iteration after the next in $A[i-2]$. From Equation 3.1, the memory reuse is 1 for $A[i-1]$ and 2 for $A[i-2]$ for the access $A[i]$. The memory reuse of $A[i-1]$ is 1 for $A[i-2]$. The reuse between $A[i-1]$ and $A[i-2]$ is considered to account for a migration between these accesses. The total MR from this statement within the inner loop is $N + N * 2 + N$, where N is the iterations of the inner loop. Hence, for the loop nest with outer loop iterations M, the MR is $M * [N + 2 * N + N]$. Similarly, the second statement contributes $M * [3 * N + 2 * N + N]$, and the third statement adds $M * [5 * N]$ to the total memory reuse of the loop nest. In other words, the total MR of the loop nest is $15 * M * N$. This reuse is generated by $3N$ unique accesses. Note that, the total accesses is still $3 * N * M$, that is $3N$ array elements accessed M times, but unique accesses are only $3N$. By substituting these accesses in Equation 3.3, UMR is $15 * M * N/3N = 5 * M * N$. The loop contains eight instructions and by substituting the values in Equation 3.4, we find memory-reuse density equal to $5 * M/8$.

3.4 PinIt Optimization

PinIt optimization focuses on determining the location where pin/unpin calls should be hoisted to reduce call overheads and transforming loops to maximize their memory reuse density.

3.4.1 Call Hoisting

The pin (pin/unpin) calls are hoisted at the outermost loops. Hoisting prevents repeated calling of pin/unpin functions when present inside a loop. However, some of these pin/unpin calls could still remain inside some inter-procedural-external loops, that is, loops in some

other function calling the function containing the pin/unpin calls. External loops are formed when the functions are not in-lined. With the use of call graphs, such external loops can be determined, and pin/unpin calls can be moved outside the external loop. Because the external loops can reside in various files such as header files and libraries, hoisting is accomplished after the entire program is linked. The pin calls can be hoisted at the outermost hoist points only after the two key goals of Pinit–to identify regions with large cache reuse and to determine the placement of pin/unpin calls–are realized. The loop, whose pin calls should be hoisted, already adheres to the two goals intra-procedurally. We must now ensure if these predicates still hold at the new hoist points surrounding the external loops. The reuse density (MRD) must be recalculated at the hoist points to check if the external loop nest has enough memory reuse to negate the loss of scheduler flexibility. We calculate unit memory reuse (UMR) and count the instructions along the path of the external hoist point enclosing several procedural calls. Using these values, we calculate the new MRD at the hoist points and then use a predicate to check if the new value exceeds the RDT (reuse density threshold).

Formally, let f_p be a function originally containing pin calls and I_p the set of instructions within f_p; f_p is called by a set of functions, $F = \{f_1, f_2....f_n\}$. F_L, which calls f_p inside a loop, is a subset of F; that is, $F_L = \{f_i, f_j....f_m\} \subset F$, and I_i is the set of instructions corresponding to the loop in $f_i \in F_L$. If MRD_{f_p} is the reuse density of the pinned loop in f_p, MR_{f_p} is the memory reuse inside f_p, and MR_{L_i} is the reuse in external loop, L_i in $f_i \in F_L$, the pin call must be hoisted outside the loop if

$$\frac{MR_{f_p} + MR_{Li}}{I_p + I_i} > RDT. \tag{3.5}$$

For functions in F/F_L, in which the calls to the function f_p are not inside a loop, the pin/unpin calls must surround the call to f_p if

$$MRD_{f_p} > RDT \quad \& \quad F_L \neq \emptyset, \tag{3.6}$$

where RDT is the "Reuse Density Threshold". The steps for minimizing pinning and hoisting pin calls are summarized in Algorithm 2.

Algorithm 2 Pinning Algorithm

procedure PINLOOP
for each Outermost loop $L \in$ *Function* F:
    $N_L \leftarrow$ Number of Iterations of L
    Memory_Reuse(MR_L) =
    Σ Memory_Reuse of Subloops(L)+ Σ(Memory_Reuse)
    $MRD_L = \frac{MR_L}{Instructions}$
    if $MRD_L > RDT$ **then**
        Pin Loop L
End for
procedure HOISTING
for each Pinned Loop $L \in$ *Function* F:
    $RemovePin \leftarrow False$
    $U \leftarrow u_1, u2, ..u_n$ s.t each u_i calls F
for each $u_i \in U$:
    if F is within Loop l_i of u_i **then**
        $RemovePin \leftarrow True$
        Recalculate MRD for l_i with F
        if $MRD > RDT$ **then**
            Hoist Pin Outside li
End for
    if $RemovePin == True$ **then**
        if $MRDof$F$> RDT$ **then**
            Hoist Pin outside non-Loop calls of F, $u_j \in U$
        Remove Pin calls from F
End for

As an illustration, in Figure 3.2, the loop in function Foo() is pinned at the pre-header and unpinned at the exit blocks of the loop, which is based on the initial intra-procedural analysis. However, inter-procedurally, function Foo() is called once in procedure Boo() and multiple times within a loop in Goo(). The repeated invocation of pin (pin/unpin) calls by Goo() causes high system overhead because of switching contexts from the user space to the kernel and back. To prevent multiple switching overhead, PinIt by using the information from the call graph will hoist the pin/unpin calls at points H1 and H2 in Goo and at points H3 and H4 in Boo if equations 3.6 and 3.5 are met, respectively. Note that in function Boo(),

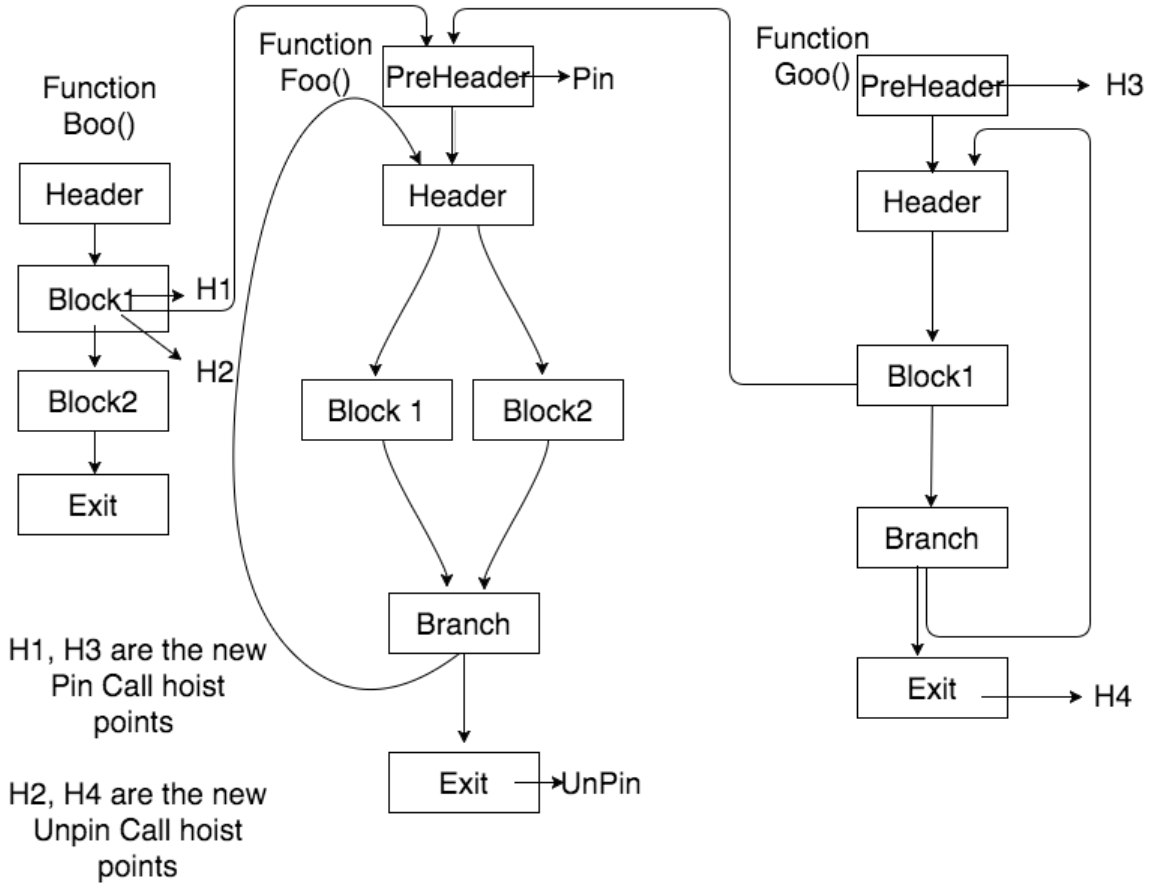


Figure 3.2: hoisting outside external loops

calls are hoisted surrounding the call for Foo(), and in Goo(), the calls are hoisted outside the loop.

3.4.2 Loop Transformation

Pinning a large loop with low memory reuse not only decreases the gain from pinning but also restricts the scheduler from using resources efficiently. To account for this overhead, we pin the loop only if the reuse density is above a reuse density threshold. Furthermore, to increase the memory reuse density of a loop, we carry out loop splitting, so that only dependent instructions containing high memory reuse forms one loop nest and the rest another. For example, splitting the above loop by removing non-memory-reuse instructions that are not dependent increases the MRD to $5M/4$, twice the previous value.

```
1 void foo(int M, int N)
2 {
3     ...
4   for(int j = 0; j < M; ++j)
5   for(int i = j; i < (j+N); i++)
6   {
7       A[i] = A[i -1] + A[i-2];
8       B[i] = B[i -3] + B[i -2];
9       C[i] = C[i-4] + k;
10      k = i + k;
11  }
12
13  for(int j = 0; j < M; ++j)
14  for(int i = j; i < (j+N); i++)
15  {
16      x = i + 2 + x;
17      y = n + i + y;
18      z = z+1
19      m = m--;
20  }
21  ...
22 }
```

Code 3.4: Function foo(), split loops

If a loop can be split in multiple ways, we use the method outlined in Algorithm 3 to select the one that maximizes the MRD.

Algorithm 3 Loop Splitting

1: **procedure** LOOP SPLITTING
2: *for each innermost loop* $L \in$ *Function* F:
3: $G \leftarrow \{g_1, g_2 ... g_n\}$s.t. g_i is minimal reuse statements
4: $S_i \leftarrow g_i \bigcup$ transitive statements
5: Invalidate the unreliazable group $g_i \in G$ which conflicts with S_i
6: Select S_i s.t. MRD_{S_i} is maximum

For example, in Code 3.3, minimal-reuse groups of statements are $g_1 = \{s_7, s_8, s_9\}$, $g_2 = \{s_8, s_9\}$, and $g_3 = \{s_9\}$. After adding transitive statement s_{14} to the groups, we find statements $s_8, s9$ are included in group g_1, so g_2 and g_3 are invalid. We remove loop-splitting groups such as g_2 and g_3 that are unrealizable. The new $g_1 = \{s_7, s_8, s_9, s14\}$ results in higher reuse density equal to $5M/4$. We split the loop into L_1 and L_2 such that L_1 includes statements $\{s_7, s_8, s_9, s_{14}\}$ and L_2 receives $\{s_{10}, s_{11}, s_{12}, s_{13}\}$ as shown in Code 3.4.

3.5 Pin Threshold

To determine the reuse density threshold value that benefits all the programs in a suite fairly, we chose to cluster the MRD values using kmeans. First, we collect MRD values of all loops of different benchmarks in a benchmark suite. We derive k means that lies within the range of MRD value of different loops of a benchmark in the suite. We start with $k = 1$ and increment k till we find a mean MRD value that is within the range of MRD values for the application. For example, when $k = 1$ we might not have any loop in a benchmark with an MRD value which is higher or equal to the mean. We increment k to find the second mean, and again if the mean is greater than the maximum MRD value in the application, we increment and repeat until we find a mean value that lies within the range of MRD values of the applications as described in Algorithm 4. The different reuse density thresholds are listed in Table 3.1.

Table 3.1: Reuse Density Threshold

Benchmark	**Suite**	**RDT**
djpeg	Mediabench	17.55
h263dec	Mediabench	22.12
h264dec	Mediabench	20.05
mpeg2dec	Mediabench	40.30
sd-vbs(all)	sdb-vbs	2.67

Algorithm 4 Reuse Density Threshold Algorithm

1: **procedure** FINDREUSEDENSITYTHRESHOLD
2: $BS \leftarrow \forall B \in BenchmarkSuite$
3: *for each bench* B *in BS*:
4: $P \leftarrow v_1, v2, ..v_n$ s.t each $v_i \in B$
5: *End for*
6: $k = 1$
7: *while True*:
8: $RDT \leftarrow k^{th}mean \in kmeans(P)$
9: *for each bench* B *in BS*:
10: **if** $RDT < max(v_i) \in B$ **then**
11: $RDT_B \leftarrow RDT$
12: Remove B from BS
13: *End for*
14: **if** $is\_empty(BS)$ **then**
15: Break
16: $k+ = 1$
17: *End while*

3.6 Pinit Runtime

For pinning efficiently, we need a dynamic optimization that restricts the number of processes that can be pinned to a processor. Pin calls are implemented using the cpu_schedsetaffinty system call in a shared library that maintains a shared bit-mask, a mask of reserved processors in the system. The library call determines if an application can be pinned by comparing the shared bit-mask to the CPU mask, a mask with the bit pertaining to the processor on which the application is currently executing is set, collected from the get_cpu system call. If the process can be pinned, the CPU-mask bit is atomically set in the shared bit-mask and any subsequent requests to the same CPU are denied. The shared library in the pinIt framework consists of four exposed functions–pininit, pin, unpin, and pinfree. Pininit function is inserted at the entry block of the main function of a program, and similarly, pinfree function is instrumented at all the exit blocks of the program. We used a shared-bit mask to maintain an account of shared processors with the help of **shm** semantics provided by Linux. Linux also provides the **cpu_sched_setaffinity** system call to pin a process and is used inside the

pin call. The average overhead of each call is listed in Table 3.2. The applications were instrumented with passes written in LLVM 3.8 and compiled with O3 optimization.

Table 3.2: Shared pin library characteristics

Lib Call	**AvgTime**
PinInit	36μs
Pin	6.6μs
UnPin	1.6μs
PinFree	34.6μs

3.7 Experiments

3.7.1 Experimental setup

Testbed: We used Linux OS running on two different Intel Xeon Processor machines, one with 8 cores and the other with 16 cores with configurations as mentioned in table 4.2 in a carefully controlled environment. Specifically, to prevent the effects from warmup, we cleared shared resources such as the cache before every run. With hyper-threading disabled, the number of processes executed in a batch was more than the number of processors in the machine. In such an environment, the machine was overloaded by 50%, that is, the number of processes scheduled was 1.5 times the number of processors. Consequently, in 8 core machine, we ran 12 processes, and in 16 core machine we scheduled 24 processes for execution. Among the processes in the load, 100% processes were of high-priority and the rest overloaded 50% was of low-priority. In other words, 8 high-priority applications and 4 low-priority applications were scheduled on 8 core machine, and 16 high-priority and 8 low-priority applications were executed on 16 core machine

Benchmarks: To demonstrate the usage of Pinit, we used Mediabench benchmark suite, a set of real-world media applications and San-Diego vision-based benchmark suite (sd-vbs). The applications were classified into higher- and lower-priority applications. Because applications such as decoders, in contrast to encoders, are on the critical path of internet traffic, they were classified into higher-priority applications, and the encoders were bundled

Table 3.3: Configuration of experiment machines

Features	**8 core**	**16 core**
Core count	8	16
CPU Base frequency	2.40 GHz	2.2 GHz
CPU Turbo frequency	2.66 GHz	3 GHz
CPU architecutre	Nehalem	SandyBridge
Sockets	2	2
Core/socket	4	8
L1 Cache	32 KB	32 KB
L2 Cache	256 KB	256 KB
LLC Cache	8 MB	20 MB
Linux kernel	4.5.0	4.5.0

into the lower-priority application set. Table 3.4 lists the configuration of different mixes of high-priority set of applications for Mediabench. We used the same set of low-priority applications consisting of encoders with sd-vbs as high-priority applications, that is, all the experiments included the encoders cjpeg, h263encode, h264encode, and mpeg2encode executing simultaneously as the lower-priority set of applications. The decoders in mediabench, and sd-vbs, which form the high-priority set, were compiled and executed with the PinIt framework. The various applications used have distinct characteristics in terms of the loops that are pinned and the execution time. The mixes of such diverse applications push the scheduler to capture various scenarios thus testing the robustness of PinIt.

Table 3.4: Mixes of high-priority applications in MediaBench

MixBench	**High Priority Set**
Mix1	djpeg h263dec h264dec mpeg2decode
Mix2	h263dec h264dec mpeg2decode
Mix3	h264dec mpeg2decode
Mix4	mpeg2decode
Mix5	djpeg
Mix6	h263dec
Mix7	h264dec
Mix8	djpeg h263dec h264dec
Mix9	djpeg h263dec
Mix10	djpeg h264dec
Mix11	djpeg mpeg2decode
Mix12	h263dec mpeg2decode

Table 3.5: Application characteristics

Pin Applications	Apptime	pins	PinTime
djpeg	0.35s	2	0.0004
h263dec	2.66s	1	2.57
h264dec	0.6s	1	0.00001
mpeg2decode	9.7s	1	9.4
disparity	5.74	78	2.87
localization	0.476	132	0.321
mser	1.08	78	0.85
sift	7.2	36	3.7
stitch	63.5	9	1.2
svm	0.4	1	0.04
tracking	1.6	1	0.005

3.7.2 Experimental goals

The process when pinned must be non-preemptive, otherwise, the cache of the pinned process will be polluted by the process that pre-empted the pinned process. The default CFS scheduler pre-empts pinned processes and is not suited for PinIt. In the function call PinInit, the scheduling for the high-priority application is changed from CFS to non-preemptive scheduling(FIFO), which prohibits preemption of the pinned process except only when an IO operation is requested by the process. The IO request can be very time consuming to skip the precious CPU cycles. Only the high-priority applications are scheduled non-preemptively. The low-priority applications are scheduled using the default CFS in PinIt. We compare PinIt against priority CFS that differentiates the high- and low-priority applications through nice values. The nice value was set to -20 (highest) for high-priority applications and default 0 for low-priority applications. In the red-black tree-based runqueue of CFS, the nice value plays a part in selecting the next application for execution. The lower the nice value, the higher the priority for an application to be picked up for execution.

3.7.3 Experimental Results

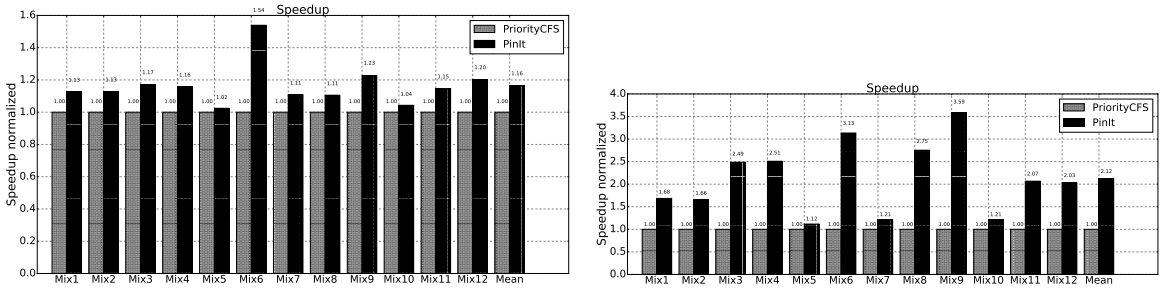
We demonstrate the speedups gained by PinIt over priority CFS schedulers. Each overloaded batch of each of the mix was run for 12 iterations with each framework – PinIt and priority CFS, to collect various results of the applications in the batch. The speedup reported in the figures is normalized to that of the priority CFS scheduler.

Table 3.6: Average batch throughput for 16threads(8threads) normalized to priority CFS

Benchmark Suite	**PinIt**
Mediabench	1.23x (1.01x)
sd-vbs	0.98x(1x)

Table 3.7: Average latency for 16threads(8threads) normalized to priority CFS

Benchmark Suite	**PinIt**
Mediabench	45% (13%)
sd-vbs	8%(19%)



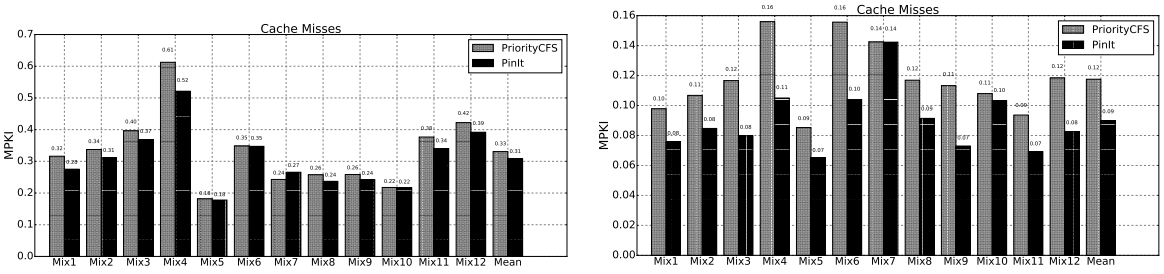
(a) 8 core (b) 16 core

Figure 3.3: Mediabench: speedup of high-priority applications normalized to priority CFS vs PinIt

Mediabench

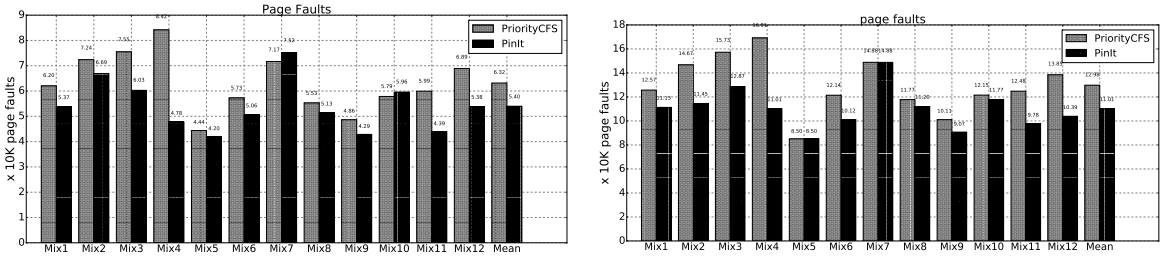
The average speedup of high-priority applications in PinIt compared to CFS priority scheduling is 16% in 8 core machine as shown in Figure 3.3a and 2.12x in 16 core machine as shown in Figure 3.3b. Although, the low-priority applications in each framework was scheduled with default CFS scheduling mechanism, scheduling decisions for high-priority applications directly affect the completion time of low-priority applications. For each framework, we

also show the completion time of the batch normalized to that of the priority CFS scheduling. In 8 core, PinIt managed to complete the batch as fast as priority CFS scheduling, and in 16 core, the ample time saved in completing the high-priority job was used in completing the low-priority jobs 16% faster as shown in Table 3.6. To demonstrate that the increase in the overall throughput of the high-priority applications is not at the expense of the latency of each application, we calculate the average latency of applications in terms of execution time for the high-priority mixes for each framework. We compare the latencies of each framework. PinIt compared to priority-CFS was faster on average by 13% in 8 core and by 45% in 16 core as shown in Table 3.7.



(a) 8 core (b) 16 core

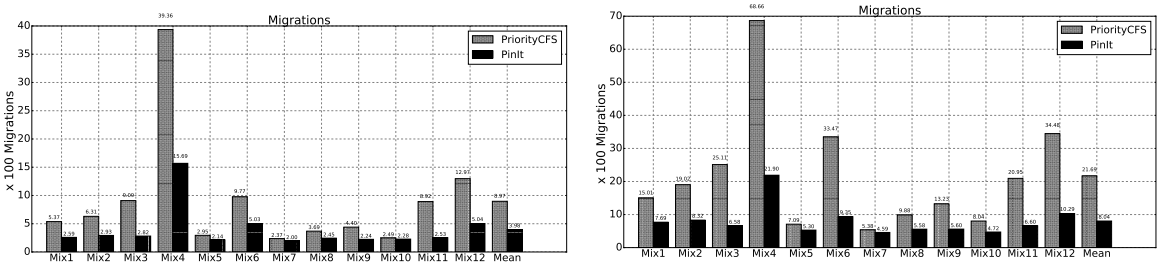
Figure 3.4: Mediabench: cache Misses (MPKI) in priority CFS vs PinIt



(a) 8 core (b) 16 core

Figure 3.5: Mediabench: page faults in priority CFS vs PinIt

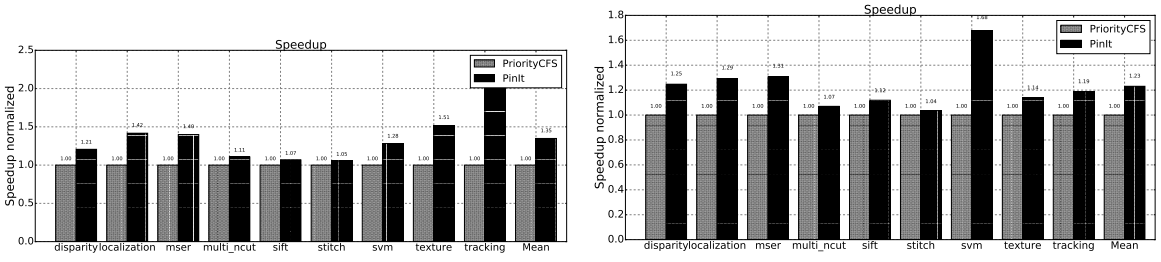
We compare the system behavior when using PinIt versus the priority CFS by reporting cache misses, page faults, and CPU migrations collected using the Perf tool. PinIt relies on avoiding dubious migrations than can lead to superfluous cache misses as well as page faults. These three factors together contribute to the speedup of the application. Migrations



(a) 8 core (b) 16 core

Figure 3.6: Mediabench: migrations in priority CFS vs PinIt

can not only increase data misses but also can result in inefficient use of CPU cycles. Cache misses and page faults directly affect the execution of an application. In NUMA machines, such as the ones used in our experiments, the cache misses because of migration from one NUMA node to another can be more drastic than intra-NUMA node migration. In 8 core, for the entire batch, on average, cache misses are reduced by 6%, page faults (minor and major together) are decreased by 17.4%, and migrations are reduced by 55.6% in PinIt as shown in Figures 3.4a, 3.5a, and 3.6a, respectively. Also in 16 core, for the whole batch, on average, cache misses are reduced by 25%, page faults are reduced by 15%, and migrations are cut down by 63% as shown in Figures 3.4b, 3.5b, and 3.6b, respectively. The stalls in PinIt are almost the same as priority-CFS.

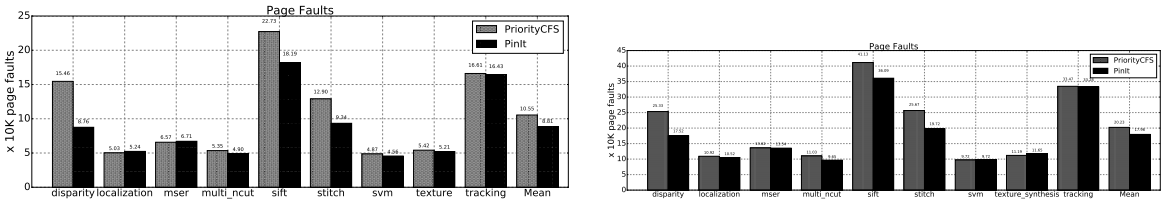


(a) 8 core (b) 16 core

Figure 3.7: sd-vbs: speedup of high-priority applications normalized to priority CFS vs PinIt

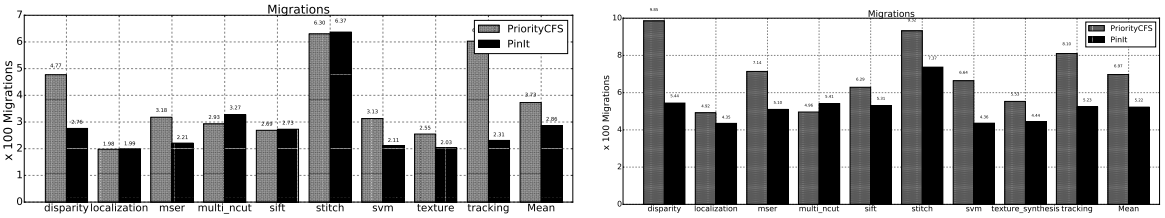
sd-vbs

The average speedup of high-priority applications in PinIt was 1.35 times that in CFS priority scheduling in case of 8 core machine as shown in Figure 3.7a, and 1.23 times in 16 core machine as shown in Figure 3.7b. In 8 and 16 core, PinIt managed to complete the batch as fast as priority CFS scheduling as shown in Table 3.6. Latency wise, PinIt compared to priority-CFS was faster on average by 19% in 8 core and by 8% in 16 core as shown in Table 3.7.



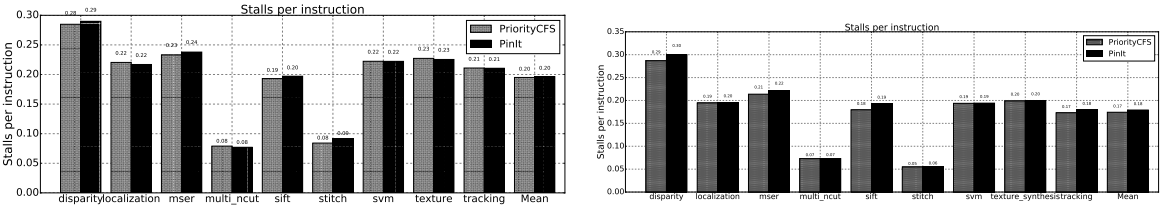
(a) 8 core (b) 16 core

Figure 3.8: sd-vbs: page faults in priority CFS vs PinIt



(a) 8 core (b) 16 core

Figure 3.9: sd-vbs: migrations in priority CFS vs PinIt



(a) 8 core (b) 16 core

Figure 3.10: sd-vbs: stalls in priority CFS vs PinIt

In 8 core, for the entire batch, on average, MPKI is not reduced, however, cache misses are fewer by a small margin for many sd-vbs applications in PinIt than priority CFS, but

PinIt memory reuse benefits manifest as TLB hits or reduced page faults (minor and major included). Note that minor page faults indicate TLB misses. In 8 cores page faults are decreased by 16.4%, shown in Figure 3.8a, and migrations are reduced by 23.3% as shown in Figure 3.9a. Also in 16 core, page faults and migrations are reduced by 11% and 25% as shown in Figures 3.8b and 3.9b, respectively. Although in PinIt cache misses do not show a considerable difference with priority CFS, fewer migrations in PinIt compared to priority CFS (as shown in Figure 3.9), avoids the unnecessary halting of a process, migrating to a new processor, and resuming the process overheads thus achieving speedup. The speedup achieved in sd-vbs can be attributed to the reduction in page faults (TLB misses) and migrations.

Experimental Analysis

Machine Utilization: Within the same amount of time PinIt completes more number of jobs in most cases except in sd-vbs 16threads (2% fewer) as shown in Table 3.6. Note that, the machine utilization cannot be compared against prior works like Bubble-flux [2] because bubble-flux starts at 50% utilization, i.e. with a job on every alternate core, and then increases machine utilization by consuming the idle cores. In our environment, there are no free cores, apart from every core being occupied by a high-priority process, any CPU cycles wasted by the high priority process is consumed by the low-priority process. In other words, the machine is overloaded to utilize all the possible cycles to give lower latency to high priority jobs, while completing the batch at faster or the same rate as priority-CFS.

Fairness. The environment in which PinIt is used consists of a set of high-priority applications equal to the number of cores in the machine and a low-priority set is run as a batch. Since each high-priority application is executed by a core in a non-preemptive manner, no high-priority application starves for processing time, and from the above results, we can safely conjure that the low-priority job takes almost same time or less to complete in PinIt as in priority CFS. The standard deviation in the execution time is reduced in PinIt because

of reduced migrations due to pinning. That is, the repeatability of the high priority applications is increased because of reduction in random behavior caused by non-deterministic migrations.

Loop Splitting: This loop transformation is useful only in djpeg, in which a loop was available for splitting after hoisting. In others, the hoisting optimization hoisted all the pin calls to inter-procedural outermost loops, thereafter Loop Splitting pass did not find any loops to split. The differences in the metrics of djpeg versions before and after loop splitting are as shown in Table 3.8. Note that djpeg has two pinned loops and the MRD of one loop was increased without change in the number of pin calls.

Table 3.8: Djpeg before and after loop splitting optimization

Metric	Before	After
MRD	13.97	17.55
Speedup	17.9%	20.9%

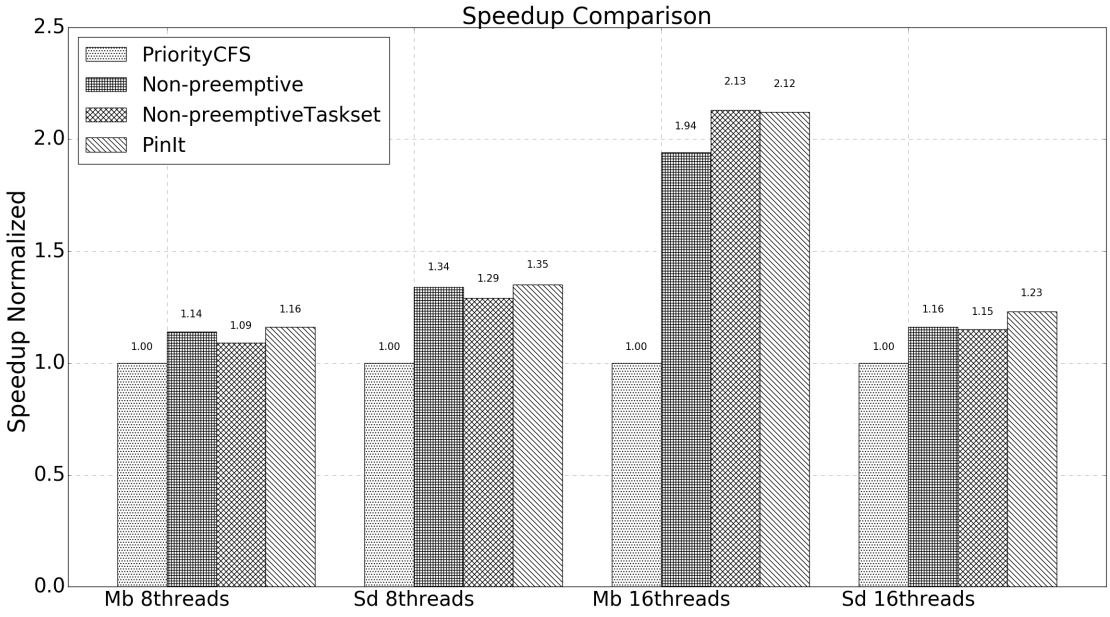


Figure 3.11: Average normalized speedup: Pinit extracts the best of scheduler flexibility and processor affinity

PinIt performance. PinIt improves the performance of the applications by reducing all or some of the factors among cache misses, page faults (TLB misses), and migrations as was

seen in mediabench in which all three factors were reduced thus improving performance phenomenally, and in sd-vbs by reducing page-faults (TLB misses) and migrations. **non-preemptive scheduling:** Although PinIt benefits from non-preemptive scheduling, PinIt outperforms non-preemptive scheduling by as much as 28% in Mediabench and by as much as 26% in case of sd-vbs. **non-preemptive affinity.** Our work proposes that scheduler flexibility is necessary and complete affinity to a processor for the entire life cycle of a process can be harmful. We observed that PinIt when compared to complete non-preemptive affinity fairs up to 16% faster in Mediabench and up to 51% faster in sd-vbs. Few applications in mediabench, however, gained higher benefits with complete non-preemptive affinity than pinning fewer areas denoted by PinIt. We observed that the PinIt missed gains mainly from file read operations that were pinned in complete non-preemptive affinity. Figure 3.11 compares PinIt with purely non-preemptive scheduling and complete non-preemptive affinity. As required, *PinIt extracts the benefits of scheduler flexibility and processor affinity as and when necessary, thus mostly performing better than both just non-preemptive scheduling and non-preemptive whole process pinning.*

3.8 Related Work

Some of the prior works have increased the machine utilization at the cost of the Quality of Service (QoS) in a high- and low-priority process environment. Some works have focused on improving machine utilization through compiler analysis and transformations. Few other works have developed scheduling techniques for efficient usage of the cache, while some have used a compiler to assist the scheduler in unique ways. However, to the best of our knowledge we found no work that uses cache affinity of the co-executing programs or determines cache-affinity dynamically to avoid harmful migrations let alone technique determined through compiler analysis for guiding the scheduling decisions. We discuss the above topics in detail.

Bubble-up [24] generates a QoS versus memory pressure sensitivity curve to statically

co-locate two applications together. ”Optimal task assignment in multi-threaded processors” [27] also determines the best static assignment of a fixed offline application set. The authors use extreme value theory for generating the best assignment of jobs. Autopin [25], an offline tool similar to Valgrind finds the best thread-to-core mapping through an offline iterative process. These problems are geared to address the problem of process co-location and not process migration like PinIt does, that is, they do not deal with allowing or disallowing process migration to reduce cache misses. Using the memory pressure curve the authors of Bubble-up developed Bubble-flux [2], in which the low-priority processes are simultaneously executed along with the high-priority processes only when the QoS of the high-priority processes does not drop. This work targets increasing machine utilization by using low-priority processes to use CPU cycles wasted by high-priority processes. [28] attempts to assign VCPU for entire execution time to a CPU based on the history of execution and does not dynamically set or reset the affinity of the VCPUs. This is the closet work that addresses the problem of migration, however in case of VCPUs and not processes in a virtual execution environment.

Improving the cache efficiency by focusing on the scheduling techniques has been studied in many works. Among them, Thread tranquillizer [29], studies several combinations of scheduling and memory allocation techniques. It uses fixed scheduling, providing a process with a non-constant time quantum for execution by classifying the process as either memory-intensive or compute-intensive, and random memory allocation to increase cache reuse. The work on schedulability analysis of the Linux Push and Pull scheduler with arbitrary processor affinities [30] shows that job-level fixed-priority scheduling with arbitrary processor affinities is more general than global, partitioned, and hybrid scheduling.

Among several compiler works that focus on process performance ”Reducing Context Switch Overhead with Compiler-Assisted Threading” [31] context-switches only in the region of execution that has a minimal amount of state to be saved by doing live register analysis. ”Region scheduling: efficiently using the cache architectures via page-level

affinity" [32] schedules the process to the appropriate processor with the cache that has the memory region by building a cache map. Works in [33, 34, 35, 36] present loop optimization techniques for improving data reuse by the loop. Data reuse optimization techniques found in the literature either perform compiler analysis to change data access patterns or utilize the runtime system and improve reuse. Few other works such as CRUISE: Cache Replacement and Utility-aware Scheduling [36] and "Reuse Distance-Based Cache Hint Selection" [37] use compiler to aid the hardware for better cache management.

None of these works to the best of our understanding solve the problem of process migration dynamically let alone *influence* the scheduling decisions of the operating system scheduler non-intrusively towards solving the problem of process migrations by the scheduler. Hence, we use CFS which has data-locality awareness in-built and also provides API's and mechanisms like taskset for absolute affinity as our baseline.

3.9 Conclusion

We developed Pinit, a technique to efficiently influence the OS scheduler to speedup the applications in an overloaded environment by inducing processor affinity. Maintaining the right processor affinity by pinning at the right moment can reduce the number of cache misses, TLB misses from process migrations. However, excessive pinning can lead to inefficient usage of resources. To achieve optimal pinning, Pinit offers a solution through compiler analysis of memory reuse and carefully balancing the sizes of the regions pinned by relying on a new metric called Memory Reuse Density (MRD) for deciding if the loops must be pinned. Further to reduce the overheads of the pin/unpin calls, we optimize the placement of the calls based on MRD. We observed that in an overloaded environment, PinIt speeds up high-priority applications in mediabench workloads by 1.16x and 2.12x and in vision-based workloads by 1.35x and 1.23x on 8cores and 16cores, respectively, on average while completing the low-priority jobs in almost the same time as priority CFS.

CHAPTER 4

BEACONS: A COMPILER - SCHEDULER INTERFACE FOR DYNAMIC ATTRIBUTES

In the last chapter, we presented the PinIt framework which utilized only the Memory Reuse Density attribute required to influence process migration. In this chapter, we demonstrate that the dynamic attribute passing from the compiler to the scheduler is a much more powerful mechanism and PinIt is a lightweight precursor to developing a more generic framework of the compiler to the scheduler information conduit called **Beacons**. With this motivation, we present a compiler-runtime framework where the compiler statically analyzes and inserts beacons into a program. At runtime, these beacons broadcast information about forthcoming resource needs to the scheduler through a common library. The scheduler can then aggregate and analyze the information across all the processes, constructing a global representation that captures the contentions, the duration of contentions, the sensitivity to the resources and take appropriate actions for efficiently managing the available resources. The beacons are inserted at the entrances and exits of the loops and are hoisted inter-procedurally. In our current implementation beacons carry dynamic program attributes such as the loop's memory footprint, the classification of memory usage as stream or reuse, and an estimate of their execution duration. Some attributes, such as memory requirements in certain code regions, can be obtained analytically directly through static compiler analysis. Other aspects, e.g. the amount of time a region executes can only be predicted by employing a learning mechanism that involves training. This information in the beacon can be augmented or replaced with other kinds of information. We use the timing, memory footprint, and loop classification information in the next two chapters of a throughput scheduler and a fair scheduler targeting overall latency. Later, in the scheduling for security chapter, we load the beacons with the information of predicted cache misses experienced by a process. In

future, the beacon information can be replaced with more interesting attributes that help solve different system problems.

4.1 Relevant Background

Several works have approached the problem of scheduling with just profile data. [38, 39] predicts the upcoming phases of the applications by using a combination of offline and online profiling. [40] uses a reuse distance model for simple caches calculated by profiling on a simulator for predicting L2 cache usage. [41] proposes a cache-aware scheduling algorithm. It monitors cache usage of threads at runtime and attempts to schedule groups of threads that stay within the cache threshold. It is not compiler-driven, nor sensitive to phase changes within threads. Several efforts have focused on the development of scheduling infrastructure for the shared server platforms [2, 24, 42, 43, 3]. A key feature of these efforts is their use of observation-based methods (i.e. reactive approaches) to establish resource contention (e.g. for caches, memory bandwidth, or other platform resources) and to further determine interference at runtime, and/or to assess the workloads' sensitivity to the contended resource(s) by profiling.

Beacons overcome major limitations of the above techniques as follows:

- Beacons use actual dynamic values such as loop bounds contrasted with profile-based or history-based/feedback-driven approaches.
- Beacons forecast the upcoming workloads as against reactive approaches.
- Beacons are based on loops and are more fine-grained and actionable than phase change detection approaches.

4.2 Beacons Framework

To calculate the beacon information, regardless of the type such as the time or the amount of memory footprint of a code region, the beacon framework has a **compilation component**,

consisting a sequence of compilation, profiling, and re-compilation steps to instrument the application. During runtime this information is conveyed to the scheduler through a library call which forms the **runtime component**.

Compilation component. The compilation component, as shown in Figure 4.1, is responsible for instrumenting beacons in an application in order to guide a scheduler through upcoming intense regions. The regions in code are analyzed at the granularity of loop nests. For the subsequent chapters, the resources under contention are the caches and memory bandwidth shared by all the processes in the machine. So along with the memory-footprint model, we show how a loop-timing model that captures the loop execution time is generated in this chapter. Though a compiler cannot always directly calculate the cache usage of a program via source-level analysis (since mappings in caches are dynamic and a complex function of co-executing references), it is nevertheless capable of calculating memory footprints of individual applications that occupy the cache. At the source level, loops can reveal a substantial amount of information about the required memory.

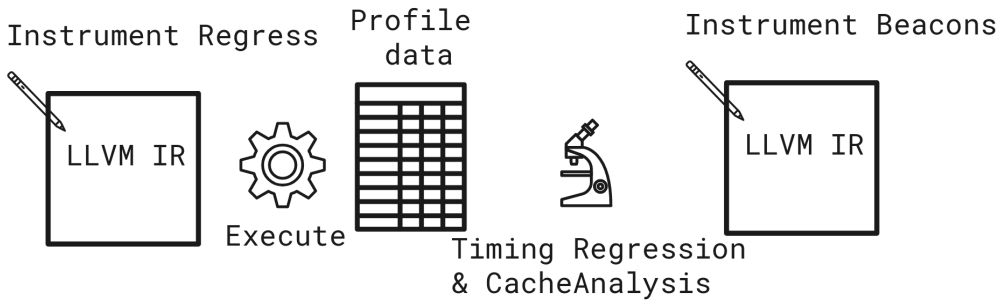


Figure 4.1: Beacon Compilation Component

During the compilation phase, the applications are first instrumented to generate profile values (loop timings for the performance schedulers in the next two chapters) through a compiler pass called the regress pass as shown in Figure 4.1. These timings are then used for training the timing predictor, a regression equation. The timing predictor is then embedded before the loop, along with two other pieces: the memory footprint equation

and the classification of the loop as either "reuse" or "streaming" based on its memory usage. The accuracy of the timing information critically depends on the structure of the loops, which is used in classifying the beacons into different types which are then treated accordingly by the scheduler, the details of which are described later in the scheduler. A beacon is inserted before the outermost loops and then hoisted inter-procedurally using the hoisting framework. At the corresponding exit of the loop for which the beacon is either inserted or hoisted, a completion beacon is inserted that indicates the end of the region to the scheduler.

Runtime component. The runtime component consists of the beacon library, which communicates with the scheduler to convey information such as timing, memory footprint coming from the beacons, the module in the scheduler that collects the beacons of multiple processes. The different schedulers in the subsequent chapters leverage this library-based beacon communication mechanism.

Next in this chapter, we will see how the timing and memory related information is generated, followed by the type of beacons and the beacon library.

4.3 Loop Timing Model Generation

We develop a linear model to predict the execution time of a loop. We learn the timing behavior of each loop independently. Thus, given a loop with various instructions, the loop time depends on the number of loop iterations, i.e. loop time is directly proportional to loop iterations.

$$T \propto N \implies T = \alpha * N, \tag{4.1}$$

where T is loop time and N is the number of loop iterations for some constant α. In a nested loop, the time depends on number of iterations of each nested loop. Once loop iterations of each nested loop are known we can claim that loop time is a function of these loop iterations,

i.e,

$$T = f(N_1, N_2, ..., N_n), \tag{4.2}$$

where T is time taken by a loop nest and $N_1, N_2, ..., N_n$ are loop iterations of the n nested loops. The loops are first normalized using the loop-simplify pass in the compiler. Loop normalization converts a loop to the form with lower bound set to zero and step increment by one to reach an upper bound. The upper bound of the loop is now equal to the number of loop iterations and can be easily extracted from the loop. Loop Time Equation 4.2 can be restated as

$$T = f(ub_1, ub_2, ...ub_n), \tag{4.3}$$

where $ub_1, ub_2, ..., ub_n$ are the upper bounds of each loop in the loop nest.

The time taken by a loop nest to execute is also equal to the time taken by each loop in the loop nest to execute the corresponding iterations individually. In other words, each instruction in the loop nest contributes to the loop time by the factor of the number of times the instruction is executed. For example, the loop in Code 4.1 will execute for the same amount of time as Code 4.2 when all other architectural influences such as cache misses are controlled.

```
1 for(i = 0; i < N; i++){
2       a = b + i;
3       for(j = 0; j < M; j++){
4           h = f - j;
5       }
6       d = c + i;
7 }
```

Code 4.1: Loop Nest 1

```
1 for(i = 0; i < N; i++){
2       a = b + i;
3       d = c + i;
```

```
4 }
5 for(i = 0; i < N; i++){
6     for(j = 0; j < M; j++){
7         h = f - j;
8     }
9 }
```

Code 4.2: Loop Nest 2

In this case, the two loop nests are also semantically equivalent and the transformation is known as loop distribution. Regardless of semantic equivalence, for timing purposes, the loops in a loop nest can be considered individually. Thus if the n loops in Equation 4.3 can be transformed into several perfectly nested loops (e.g. Code 4.1 converted to Code 4.2) then one can write the timing equation as follows:

$$T = g_1(ub_1) + g_2(ub_1 * ub_2) + ... + g_n(ub_1 * ub_2 * ... * ub_n). \tag{4.4}$$

By referring to Equation 4.1, Equation 4.4 can be rewritten as

$$T = c_1 * ub_1 + c_2 * (ub_1 * ub_2) + ... + c_n * (ub_1 * ub_2 * ... * ub_n). \tag{4.5}$$

Equation 4.5 is a linear equation in terms of each loop bound ub_k. Therefore, we use linear regression to learn the coefficients for each of the loop bounds in the loop nest and generate Equation 4.6.

$$T = c_1 * ub_1 + c_2 * (ub_1 * ub_2) + ... + c_n * (ub_1 * ub_2 * ... * ub_n) + c_0, \tag{4.6}$$

where c_0 is the constant term in linear regression.

For regression data, an LLVM pass instruments the outermost loop with start clock and stop clock directives to collect the overall time. The loop bounds of each loop in the loop nest is extracted and its relation with time is learnt using R [44]. The loop time can

increase with cache misses and other architectural factors. The increase in loop iterations will increase cache misses and hence the time. Thus, the cache misses are accounted in the time curve with multiple inputs and the time curve is faithfully learnt through loop bounds, hence generating a model which is accurate on a given architecture. The coefficients are then used in Equation 4.6, which is embedded in the application before the loop to calculate the estimated time at runtime.

4.4 Memory Footprint Analysis

Along with estimated loop timings (which indicate how long a given loop execution phase will last), analysis of memory footprint (size and type) is required for effective scheduling. The footprint indicates the amount of cache that will be occupied by a loop. The Memory footprint analysis consists of two parts. One calculates the memory footprint of the loop, and the other classifies a loop as a reuse-oriented loop or a streaming loop, which exhibits little or no reuse.

4.4.1 Calculating Memory Footprint of a Loop

For a given loop, its memory footprint is estimated based on polyhedral analysis, which is a static program analysis performed on LLVM intermediate representation (IR). For each memory access statement in the loop, a polyhedral access relation is constructed to describe the accessed data points of the statement across loop iterations. An access relation describes a map from the loop iteration to the data point accessed in that iteration. It contains three pieces of information: 1) parameters, which are compile-time unknown constants, 2) a map from the iteration to array index(es); and 3) a Presburger formula describing the conditions when memory access is performed. Generally, parameters contain all loop-invariant variables that are involved in either array index(es) or the Presburger formula, and the Presburger formula contains loop conditions. We currently ignore if-conditions enclosing memory access statements; we thus get an upper bound in terms of estimation of

the memory footprints. For illustration, list 4.3 shows a loop with three memory accesses, with two of them accessing the same array but different elements. A polyhedral access relation is built for each of them. The polyhedral access relation for $d[2 * i]$ is:

$$[N] \rightarrow \{[i] \rightarrow [2 * i] : 0 <= i <= N\} \tag{4.7}$$

where $[N]$ specifies the upper-bound of the normalized loop. It is a compile-time unknown loop invariant since its value is not updated across loop iterations. $[i] \rightarrow [2 * i]$ is a map from the loop iteration to the accessed data point (simply array indexes). $0 <= i <= N$ is the Presburger formula with constraints about when the access relation is valid.

```
1 for ( int i = 0 ; i <= N; ++ i ) {
2       ... = a[i+3];
3       d[2*i] = ...;
4       d[3*i] = ...;
5 }
```

Code 4.3: Memory Footprint Estimation Example

Based on the polyhedral access relations constructed for every memory access in the loop, the whole memory footprint for the loop can be computed leveraging polyhedral arithmetic. It simply counts the number of data elements in each polyhedral access relation set and then adds them together. Instead of a constant number, the result of polyhedral arithmetic is an expression of parameters. For $d[2 * i]$, its counting expression generated using polyhedral arithmetic is:

$$[N] \rightarrow \{(1 + N) : N >= 0\} \tag{4.8}$$

Therefore, as long as the value of N is available, the memory footprints of the loop can be estimated by evaluating the expressions. For statements that access the same arrays,

e.g. $d[2 * i]$ and $d[3 * i]$, a union operation will first be performed to calculate the actual number of accessed elements as a function of compile-time unknown loop iterations and instrumented in the program. This function is evaluated at runtime to get the exact memory footprint.

4.4.2 Classifying Reuse and Streaming Loops

A loop that reuses memory locations over a large number of iterations (large reuse distance) needs enough cache space to hold its working memory, and a loop that streams data which is reused in the next few iterations requires almost no cache space at all. For efficient utilization of cache, the scheduler must know whether a loop is streaming or not. We classify the loops using Static Reuse Distance (SRD), defined as the number of possible instructions between two accesses of a memory location. For example, in Code 4.4 the SRD between instructions 2 and 3 is in the order of $m * 3$ because the access in instruction 2 has to wait for m instructions within the inner loop to cover the distance of three outer iterations between successive access of the same memory location. The SRD between instructions 11 and 12 is in the order of two, because the same memory is accessed after two iterations.

```
1 for(i = 0; i < n; i++){
2       a = A[i-3];
3       b = A[i];
4       for(j = 0; j < m; j++ ){
5             a += B[j-1];
6             b += B[j];
7       }
8 }
9
10 for(j = 0; j < m; j++ ){
11      a += C[j-2];
12      b += C[j];
```

```
13 }
```

Code 4.4: Static Reuse Distance Example One

Any loops with a constant SRD, that is the distance between the accesses is covered within a few iterations of the same loop (e.g. the one between instructions 11 and 12 in Code 4.4), can be classified as streaming, because the memory locations must be in the cache for only a few (constant) iterations of the loop which is highly unlikely to be thrashed. More specifically, an SRD that involves an inner loop (e.g. the one between instructions 2 and 3 in Code 4.4) or outer loop (e.g. between instructions 5 and 6 in Code 4.4) where a cache entry must wait in the cache throughout the entire loop that it is dependent on – such loops are classified as reuse. Array B must be in the cache for the entire outer loop. Thus, we classify such loops in which the SRD is dependent on either an outer or inner loop as reuse loops (reuse distance here is a function of normalized loop bound N, for example), and we classify the remaining loops (with small and constant reuse distance) as streaming loops.

4.5 Beacons

The loop timing equations learnt by regression, along with the memory footprint calculations, and classification of memory reuse type are inserted before the corresponding loop/loop nest. These instructions are evaluated at runtime to predict the loop execution time and calculate the memory footprint. This information is sent to the scheduler through a library interface that communicates with the scheduler. We refer to these function calls to the library as "beacons".

4.5.1 Beacon Classification

Beacons are classified into different types based on the precision of the information being sent. The imprecision mainly arises because of different types of loops.

1. **Expected Beacon -** *Non-fixed trip count loops or Unknown loops*: Some loops have non-affine loop steps or the loop bound of the loop waits on a condition to be true. In such cases in which the loop iteration trip count cannot be found at compile time even symbolically is classified as Expected Beacon loops. An example of such a loop is:

```
1 while(bfound == false)
2 {
3     if(a[i] == i)
4         bfound = true;
5     i = i+1;
6 }
```

For such loops, during the regression runs, the loop iterations with different test inputs are also recorded, and an average expected loop bound is calculated which is then used in Equation 4.6 and memory footprint analysis. Equation 4.6 for this loop is equal to

$$T = c_1 * E + c_0, \tag{4.9}$$

where E is an expected loop bound.

2. **Precise Beacon -** *Exact time, exact memory footprint loops*: The loop bounds of all the loops in the loop nest are affine with outer loop index variables. Loops such as rectangular and triangular loops belong to this class. The loop bounds may be unknown at compile time but are derived from static variables. The following loop is triangular:

```
1 for(int i = 0; i < N; ++i)
2 {
3   a[i] = i+1;
4   for(int j = 0; j < i; ++j)
5       a[j] = a[j+1];
6 }
```

Equation 4.6 for this loop is equal to

$$T = c_1 * \frac{N^2}{2} + c_2 * N + c_0. \tag{4.10}$$

For both types of beacons, the memory footprint is calculated based on the available loop iterations information, which is either expected or precise values. The reuse classification is independent of the loop bound information.

4.5.2 Beacon Hoisting

The beacon insertion compiler pass ensures that the beacons are hoisted at the entrances of outermost loops intra-procedurally. However, inter-procedural loops can overload the scheduler with many beacon calls. Hence, if the beacons are inside inter-procedural loops, then they are hoisted outside the inter-procedural loops and also above the other call sites that are not inside loops along all paths. To hoist the beacon call, the inner loop bounds may not be available (or live) at the outermost points inter-procedurally. We use expected loop bounds of the inner loops to calculate the beacon properties, memory footprint and timing information. Unfortunately, such a conversion transforms many precise beacons to expected beacons. Note that, hoisting is a repetitive process that stops once no beacons are inside inter-procedural loops.

4.5.3 Beacon Insertion

The equations with the coefficients and loop bounds are instrumented at the preheader of the corresponding loop nests, followed by the memory footprint calculations. The variables that hold the timing and memory footprint values along with loop-type (reuse or streaming) and beacon type are passed as arguments to the beacon library call. Facilitated by the beacon library, the instrumented call fires a beacon with loop properties to the scheduler. We use shared memory for the beacon communications between the library and the scheduler. When

Table 4.1: Overhead of Beacon Library Calls

Function	Execution Time
Beacon_Init	40μs
Beacon	0.4μs
Loop_Complete	0.2μs

inter-procedural inner loops are hoisted outside a loop that already has a beacon, a newly merged beacon that accounts for the inner beacons is created. The new merged beacon time model is the same as the original outer beacon because the loop execution time is still the same. The memory footprint of these inter-procedural loops are combined and the beacon is classified as reuse even if a single reuse loop is present in the inter-procedural loop nest.

For every beacon, a loop completion beacon is inserted at either the exit points of the loop nest or after the call site for beacons hoisted above call site. The completion beacon sends no extra information other than signaling the completion of the loop phase and provides a reference to correct the scheduling actions in cases when the corresponding beacon information was imprecise. Every beacon process calls the library with three different calls – Beacon_init, Beacon, and Loop_Complete. Beacon_Init is called once for every beacon process to setup the beacon communication mechanism and takes about 40μs. Beacon and Loop_Complete calls are hoisted at every entry and exit point of the loop, respectively. These calls are very lightweight and complete in a fraction of a microsecond as shown in Table 4.1.

4.6 Evaluation

We test the efficacy of our framework on ARM-based ThunderX machines. Note that the timing information is independent of the architecture. For the subsequent chapters involving scheduling for performance, we demonstrate the efficiency of the scheduler on these machines with configuration as shown in Table 4.2.

Table 4.2: Configuration of machines used for experiments

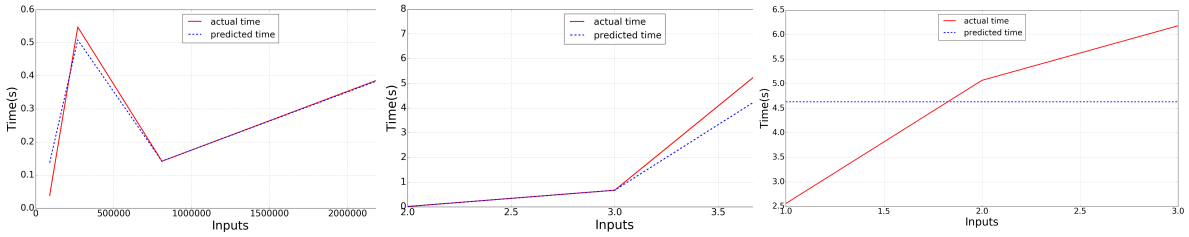
Features	**ThunderX**	**ThunderX2**
Core (thread) count	48	224
BogoMIPS	200	400
Sockets	1	2
Last-level cache	16 MB	32 MB

4.6.1 Benchmarks

A server can be training several models over streams of incoming data, or classifying data, or running thousands of physics simulations, all with the sole goal of maximizing throughput. To demonstrate the beacons work on such requirements, we used Polybench (v4.2.1). It consists of linear algebra, matrix multiplication, and solver kernels, mainly used in machine learning and scientific computations. We augmented this with Rodinia (v3.1). It consists of several structure grid, graph traversal, dynamic programming, linear algebra, and spectral method kernels from medical imaging, bioinformatics, and biological and physics simulations. Together, these suites encompass a wide swath of server functionality. For instance, several of Rodinia's benchmarks have functional overlap with CloudSuite [45]. CloudSuite's "data-analytics" benchmark performs classification (naive Bayes) and "in-memory-analytics" benchmark runs a collaborative filtering algorithm (alternating least squares). These benchmarks can be replaced with Rodinia's backpropagation benchmark (backprop), which is fundamental for many classification algorithms, and nearest neighbors benchmark (nn) which can be used for collaborative filtering and recommendation systems.

We run experiments and report all the benchmarks in Polybench. In Rodinia, we did not consider matrix and lud benchmarks as these are already covered in Polybench. Mummergpu in Rodinia is a cuda kernel based benchmark and we skip it.

All benchmarks in Polybench come with five discrete input sets– mini, small, medium, large, extralarge. We use three sets – small, medium, extralarge – for training the timing models. The large input set is used for testing the timing accuracy. The Rodinia suite entails benchmarks with both continuous and discrete inputs. With continuous inputs, the accuracy



(a) Backprop continuous inputs (b) 2mm discrete inputs (c) CFD (Expected Loop Bound)

Figure 4.2: Timing Accuracy of differnet types

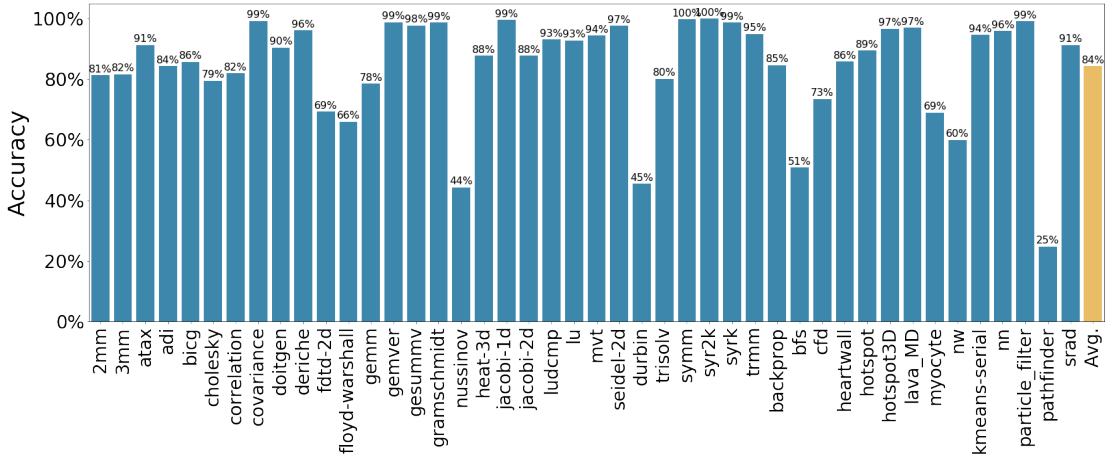


Figure 4.3: Average Timing Accuracy

of timing analysis is higher because diverse and copious inputs can be generated for training. For discrete inputs training and testing is similar to Polybench.

Both the schedulers in subsequent chapters show the efficacy of the respective solutions on these benchmarks with some specific differences in compilation which are noted in the corresponding chapters.

4.6.2 Timing Accuracy

The accuracy of the timing information ultimately depends on the types of loops and the number of inputs available for training (Equation 4.6). The expected beacons always predict a time that is based on the constant expected loop bound of the loop obtained during the regression runs. However, the actual timing curve itself can be above and below the predicted constant curve. One such curve is shown in Figure 4.2c. In CFD, the beacons that predict constant values are generated by hoisting the expected values of inner loops to a point above

their inter-procedural outer loop. As we can see, the "unknown" nature of these loops lends itself to an unreliable prediction. There is a "window" of inputs where the prediction (flat across inputs) tends to be accurate, and outside of that window the prediction is less helpful. When the loop exits, of course, an end-of-loop beacon will fire, and this helps the scheduler correct course on mispredicts.

In the case of precise beacons, the accuracy of timing info mainly depends on training. If the training inputs are of continuous form, that is, the inputs can be monotonously increased or decreased, then the coefficients learnt during regression are very precise. For example, Backprop takes one continuous integer as input and the predicted curve matches closely with completely different testing inputs as shown in Figure 4.2a. If, however, the training inputs are discrete, i.e. if only a few legal inputs are provided with the application, then the training is dependent on how representative the inputs truly are. Some benchmarks like hotspot in Rodinia have five inputs (four used for training) that capture the behaviour of the loop. The precise loop curve overlaps almost exactly with the actual time curve, similar to Backprop. In contrast, a few cases had training inputs that are not sufficient enough to generate precise coefficients, thus decreasing the accuracy. For example, in 2mm, shown in Figure 4.2b, the predicted curve deviates for the fourth input (which is the test input) by 19%. Note that the discrete inputs in Figure 4.2b are numbered 1, 2, 3, and so on. The overall timing accuracy of both expected and precise beacons together was 84%, as shown in Figure 4.3. Ultimately, the few cases of expected predictions (with low accuracy) are still manageable mainly due to loop complete beacons and performance monitoring.

This timing model, memory footprint, loop classification and the beacon framework is used in the next two subsequent chapters that develop a scheduler targeting throughput in throughput computing paradigms and a more general latency oriented fair scheduler, respectively.

Before we move on to the use of the beacons framework in developing schedulers targeting different computing paradigms such as throughput and latency, we look at the

scheduling works that use the information either collected through a compiler or by profiling the application.

4.7 Related Work

Compiler-driven approach towards understanding the program behavior have a similar goal of managing performance and interference on shared platforms but follow different approaches than what beacons do. Conservative scheduling [46] presents a learning-based technique for load prediction on a processing node. The load information at a processing node over time is extrapolated to predict load at a future time. Task scheduling is done based on predicted future load over a time window. [47] proposes a compiler-driven approach to reduce power consumption by inserting statements to shut down functional units in areas of a program where no access to the units happen. [4] is also a compiler-based technique for load-balancing and proposes early notifications before loops and considers only floating-point and mem-op instructions within loops for predicting resource usage, and places more stress on the inter-procedural placement of notifications in cluster nodes. It does not take advantage of exact analysis, multiple classes of loops, or include a regression step for further precision. It is designed for problems of load balancing at the cluster level.

4.8 Conclusion

In this chapter, we developed a generic framework for communicating the process's dynamic attributes to the scheduler. The compiler inserts models for process attributes above the loop regions in the program. These models are generated through profile data-based regression analysis and static analysis. We developed a regression-based timing model and a static memory model. The timing model on average is 84% accurate over Polybench and Rodinia benchmarks. Beacons are used in subsequent chapters for solving the problem of co-scheduling, co-location, and secure-execution. For solving co-scheduling and co-location problems, we also use the same models developed here.

CHAPTER 5
MAXIMIZING THROUGHPUT THROUGH BEACON BASED CO-SCHEDULING

In this chapter, we focus on the first use of beacons for maximizing the throughput through co-scheduling, which focuses on what processes must co-execute together and when. The placement of the co-executing processes on to the exact core and processor is left to the co-location problem. With the beacons framework in hand, now we employ it to develop a scheduler that maximizes throughput in a throughput computing setting (as outlined in [13]), which emphasizes the overall work performed over a fixed period as opposed to how fast a single or individual process completes. Throughput computing consists of a large number of jobs and the outcome of the throughput can be decided by the processes that are co-executing together. If all the processes that have conflicting resource usage execute together while some others with complementing resource requirements are waiting to be co-scheduled, the throughput would suffer drastically. And as noted earlier, these requirements do not just vary across applications but also within them. So static approaches that find a static fixed schedule or co-location such as bubble-up [24] do not suffice. Also as motivated earlier, reactive techniques that depend on hardware-performance counters to determine the resource requirements are plagued with the inability of finding the exact requirements without delays and noises. To overcome these limitations, we develop Beacon Enabled Scheduler (BES) that caters to the throughput computing paradigm by leveraging the compiler-based runtime framework beacons proposed in the previous chapter. BES uses the beacon information to dynamically decide on the applications that must be co-scheduled together so that the resource contention between the executing processes is minimized.

5.1 Relevant Background

Many works do not view the problem of co-scheduling as a separate problem and try to find the best co-location while minimizing the co-scheduling aspect. While co-scheduling deals with what and when processes must be scheduled together, co-location is concerned with the right placement of the processes, that is, what and where processes must be placed together. If both problems are solved together than one of the problems is solved sub-optimally. While optimally solving the problem of co-scheduling maximizes throughput, co-location minimizes latency. In different settings, several works have aimed at maximizing throughput. For example, [16, 17] improved throughput through load-balancing mechanisms in parallel programs and [18, 19, 20] in distributed systems. Bubble-up [24] for a fixed set of applications, first measured the QOS to memory pressure sensitivity curve offline and used the curves to decide on the co-location. Bubble-flux [2] then used the curve developed in bubble-up to perform co-location of high-priority processes and used reactive counters to perform co-scheduling of low-priority processes in order to improve machine utilization. Bubble-flux used IPC degradation to note phase changes in high-priority applications and de-scheduled low-priority applications. BES uses beacon information to decide what processes must be co-scheduled and when must these be co-scheduled together and maximizes the throughput of the system.

5.2 Beacon Enabled Scheduler (BES)

The beacon information sent by the applications is collected by the scheduler to devise a schedule with efficient resource usage as shown in Figure 5.1. The scheduler arbitrates the co-executing processes to maximize concurrency while simultaneously addressing the demand on the shared resources such as caches and memory bandwidth. Beacons can be used for different types of resource management, but in this work we only focus on efficient cache and memory bandwidth usage to improve job throughput. The beacon scheduler

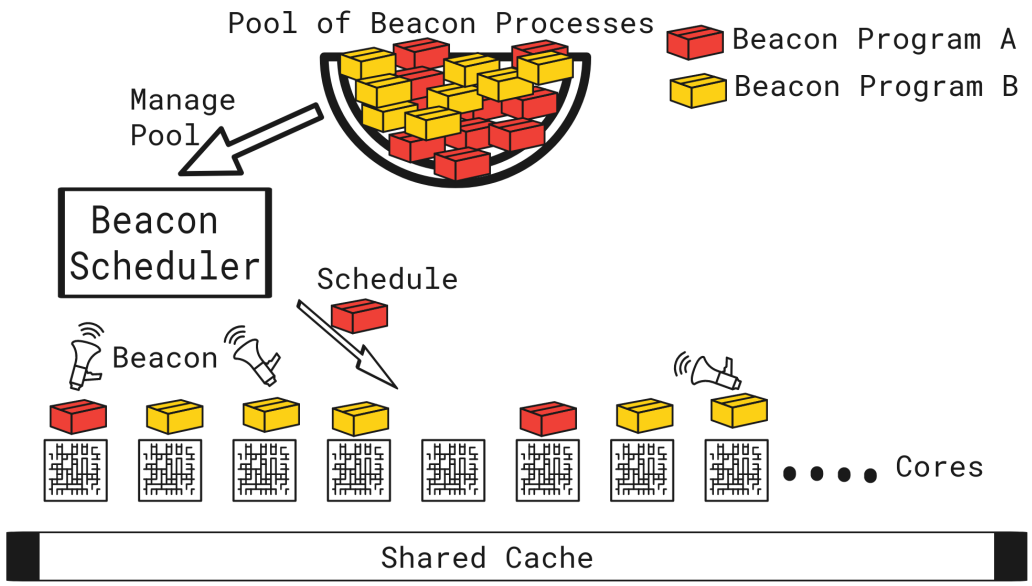
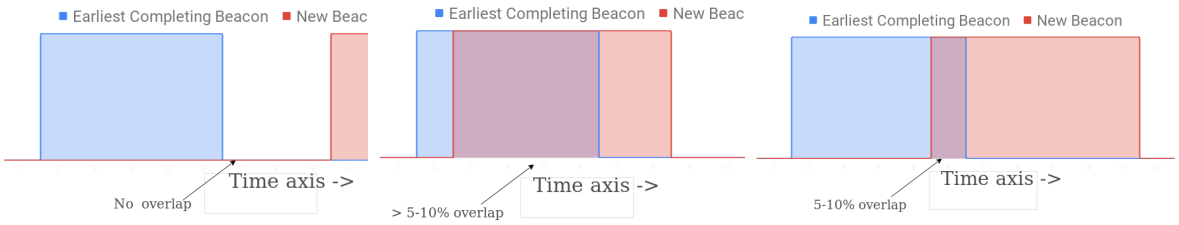


Figure 5.1: Beacon Enabled Scheduler (BES) listens to beacons to co-schedule processes

dynamically operates in two modes for the two types of loops, i.e. reuse or streaming mode as shown in the Mealy machine Figure 5.3. The two modes corresponding to two types of loops removes the adverse effects caused by multiple loops of both types executing together. The scheduler starts without a mode by launching many processes to fill up all the processors (cores) in the machine. One primary objective of the scheduler is to never idle the processors. The scheduler enters one of the two modes based on the first beacon it collects. Until a beacon is fired or after the loop complete beacon is fired, the process is treated as having no memory requirement and is referred to as a non-cache-pressure type process. All processes that do not fire a beacon are of non-cache-pressure types for their entire life cycle. During the non-cache-pressure phase, the processes have memory footprint much lower than the size of the private caches and do not harm the shared caches unlike streaming or reuse type with cache requirements exceeding the private cache.

In both modes, the scheduler acts similarly on timing information. Note that, although the loop time is trained with no other simultaneous processes, since the the scheduler avoids contention among the processes the timing of the loop must still be similar even with



(a) No Overlap (b) Big Overlap (c) Allowed Overlap

Figure 5.2: Different timing Scenarios of Incoming Beacon

multi-tenancy when scheduled by the beacon scheduler. The scheduler keeps track of the predicted earliest completing loop based on the timing information that it has gathered from the beacons. When any process fires a beacon, one of the three cases shown in Figure 5.2 can happen. In the first case, the earliest completing beacon and the incoming beacon do not overlap (Figure 5.2a). The completing beacon will relinquish its resources and are accordingly updated by the scheduler. The incoming beacon is then scheduled based on resource availability.

In the second case, they overlap for greater than 5-10% (configurable) of the beacon execution times and if the resource required by the incoming beacon is more than what is available, then the incoming beacon process is de-scheduled and replaced with another process. In the third case in which the overlap is less than 5-10%, if the incoming beacon's resource requirement is satisfied on completion of the earliest beacon process, then the process is allowed to continue but with performance counter monitoring turned on and if the IPC of the beacon processes degrade then the incoming beacon process is de-scheduled. In each case, the resource is either last level cache in reuse mode or memory bandwidth in stream mode. Also if the information is known to be imprecise (expected beacons), then the scheduler turns on performance counters to rectify its actions. Note that the loop belongs to one application or process, hence loop, process, and application may be used interchangeably.

Reuse Mode. The goal of the scheduler in reuse mode is to effectively utilize the cache by minimizing the execution overlap between the processes that are reuse bound and maybe

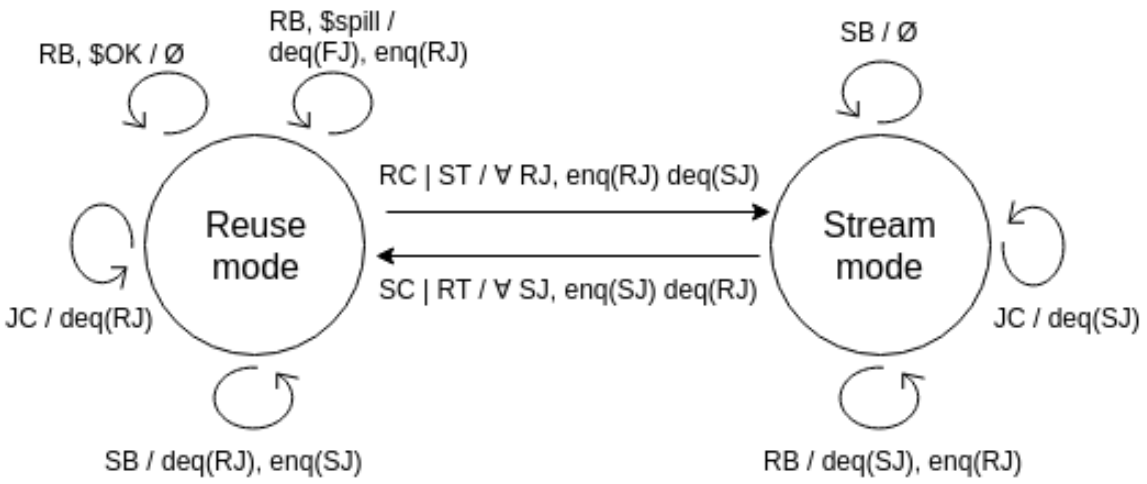


Figure 5.3: A simplified Mealy state machine of the beacon scheduler. Key: **B**eacon, **R**euse, **S**treaming, **F**iller, **J**ob, **C**omplete, **T**hreshold, **deq**ueue, **enq**ueue, **$**(cache)

pressurizing the shared cache by exceeding the capacity. At any given time in reuse mode, the cores in the machine maybe executing a mix of reuse loops (RJ) that fit the cache and non-cache-pressure (FJ) applications only as shown in the scheduler mealy machine Figure 5.3. If any of these non-cache-pressure process (FJ) fires a reuse beacon (RB), the scheduler first uses the memory information in the beacon to check if the beacon fits in the available cache space. If it does, the scheduler allows the process to continue. If the beacon is an Expected Beacon then the scheduler starts monitoring the IPC of all executing beacon processes (because only a beacon process in current reuse mode are effected from a new cache intensive process) using performance counters. Thus the scheduler takes appropriate actions based on the credibility of the beacon information.

Once the reuse loop completes (known via a loop completion beacon), the process is classified as a non-cache-pressure type (FJ) and any performance counter monitoring is stopped. If a non-cache-pressure application (FJ) fires a streaming beacon (SB), then the process is suspended and replaced by a suspended reuse process that fits in the cache. If no such suspended reuse process exists, then a non-cache-pressure process is scheduled. When all reuse loops are completed (RC) or the number of suspended stream loops hits a threshold (ST), typically 90% of the number of cores in the machine, then the remaining

reuse processes (RJ) are suspended if any, and all the streaming processes (SJ) are resumed and the scheduler switches to stream mode.

Stream Mode. A stream loop does not reuse its memory in the cache and hence is not disturbed by other co-executing processes as long as the memory bandwidth is sufficient. The expected (mean) memory bandwidth of a streaming loop can be calculated by using the memory footprint and the timing information as

$$MeanMemoryBandwidth(\mu_{bw}) = \frac{MemoryFootprint}{LoopTime} \tag{5.1}$$

In streaming mode the scheduler schedules the streaming loops (SJ) by replacing all other processes (RJ and FJ) as long as the Total Mean Memory Bandwidth (Tμ_{bw}) is less than the memory bandwidth of the machine. The memory bandwidth for ThunderX is 39.6 GBps [48] and ThunderX2 is 251 GBps [49]. Any remaining core can only be occupied by a non-cache-pressure process (FJ) because a reuse process (RJ) will get thrashed by the streaming applications. If a streaming loop completes, then it is replaced by a suspended streaming process when memory bandwidth is available. Otherwise, the process is allowed to continue as long as it does not fire a reuse beacon (RB). In other words, any non-streaming, non-cache-pressure application firing a reuse beacon is suspended and replaced by either a suspended streaming process or a non-cache-pressure application. When the number of such suspended reuse processes hits a threshold (RT), which is typically 10% of the number of cores in the machine and based on whether the reuse processes can fill the cache, the scheduler switches from stream mode to reuse mode.

An execution scenario is possible in which all streaming processes get suspended, all reuse processes are run, then after suspending more streaming jobs, all streaming processes are scheduled again in a batch, and so on. Note that during all the scheduling actions, the scheduler always maintains the invariant to keep all the cores busy. If the scheduler cannot find enough reuse or non-cache-pressure processes in reuse mode, or similarly enough

stream or non-cache-pressure processes in stream mode, then the scheduler can switch to default CFS mode, where it first fills the cores with reuse processes, followed by stream processes, until it completes the workload.

5.3 Experiments

The experiments were conducted on ThunderX and ThunderX2 machines (see Table 4.2). ThunderX socket consists of 48 processors and 16 MB L2 (LLC) cache and ThunderX2 socket consists of 128 processors and 32MB L3 (LLC) cache.

We initially sanity checked the cache pressure on ThunderX by slowly incrementing the number of active cores with one cache-hungry process each. We observed anomalous behavior once we included the second socket, though: The cache pressure was dropping instead of increasing. We did not observe this on ThunderX2, but for consistency, we used single socket on both machines. We decided to carry out experiments with 40 processors on ThunderX. In the case of ThunderX2, we conducted our experiments with 128 processors. We leave out a few processors for other system applications to run smoothly and not interfere with our results. Linux is the underlying operating system for these machines, and we used CFS in the Linux kernel version 4.15 as the baseline scheduler. The throughput environment consists of more than 200,000 jobs. CFS cannot schedule such a huge number of processes at once. When we deployed more than 5000 simultaneous processes, CFS crashed, and even at 2000 processes, CFS thrashed severely. In order to remedy this limitation of CFS, we devised a CFS batch scheduler. It starts by scheduling a task on each of the active cores, and as a process finishes, a new process is scheduled from the batch. This avoids thrashing and batch CFS can complete any number of jobs in this fashion.

We use the benchmarks Polybench and Rodinia as mentioned in the previous chapter. Specifically to BES, we set the memory footprint and loop timing threshold, only above which a beacon is fired. Because the L1 data cache size is 32KB, beacons were fired only if the memory footprint is above 32KB and also only if the predicted time is above 10ms to be

of any importance in scheduling because on average, the processing time of loop complete, reuse, and stream beacons are 116μs, 427μs, and 292μs, respectively. Leukocyte in Rodinia is one such benchmark with all beacons statically removed because the expected memory footprint is lower than 32KB and hence we do not report the values here. Streamcluster crashed with our pass and needs further investigation, hence we could not run it.

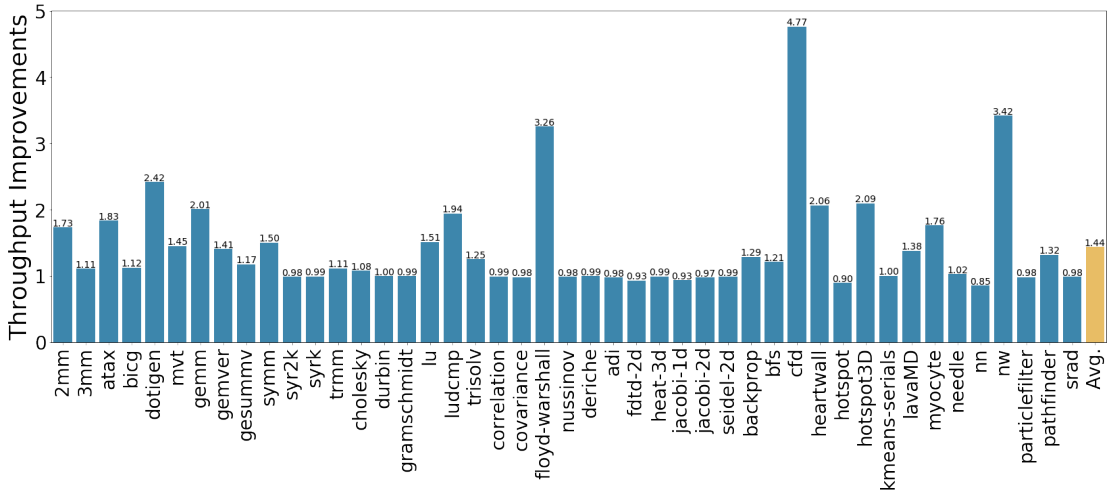


Figure 5.4: Throughput Normalized to CFS on ThunderX

Throughput. The throughput of the system is calculated as the total time required by the scheduler to finish a fixed number of incoming jobs which is same as the average number of jobs completed in the unit time when normalized with a baseline. The throughput of beacon scheduler normalized with CFS on ThunderX and ThunderX2 are presented in Figure 5.4 and Figure 5.5, respectively. On average, we achieved 44% speedup on ThunderX and 90% speedup on ThunderX2. Among 45 evaluated benchmarks from Polybench and Rodinia, we improved throughput for 27 of them on ThunderX and 38 of them on ThunderX2. The improvement is mainly attributed to the reduction in memory contention. As shown in Figure 5.6, we reduced the number of main memory accesses by 23.7% on average, with up to 76% reduction for $symm$.

Applications Adi, Fdtd-2d, Heat-3d, Jacobi-1d, and Jacobi-2d have very low reuse, and the beacon scheduler does not do anything much differently than CFS. The Deriche benchmark has only streaming loops, but alternate loops reuse the data. Thus, the streaming

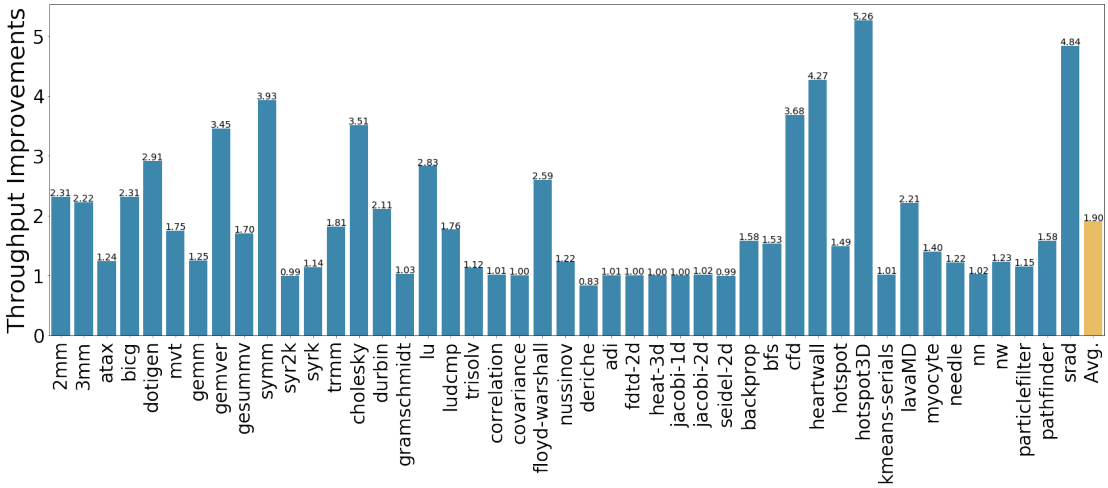


Figure 5.5: Throughput Normalized to CFS on ThunderX2

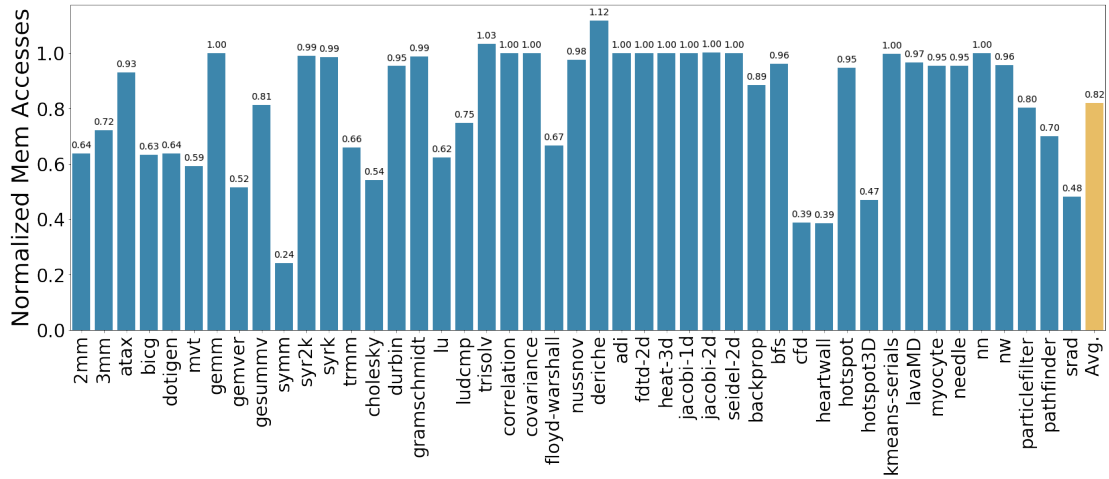


Figure 5.6: Memory Accesses Normalized to CFS on ThunderX2

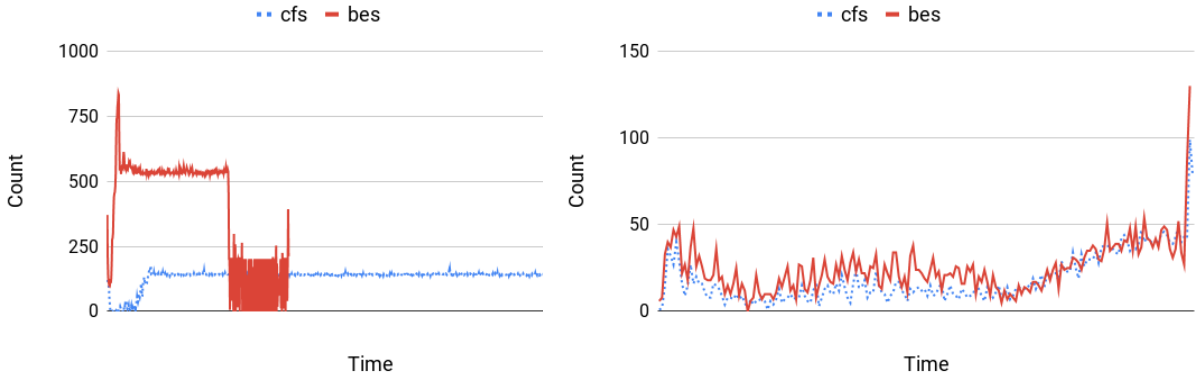


Figure 5.7: Histograms for the job completion times of CFS and the beacon-enabled scheduler (BES) for Cholesky (left) and correlation (right). The X-axis represents discrete timesteps, and the Y-axis is a count of the number of jobs that completed within a given timestep.

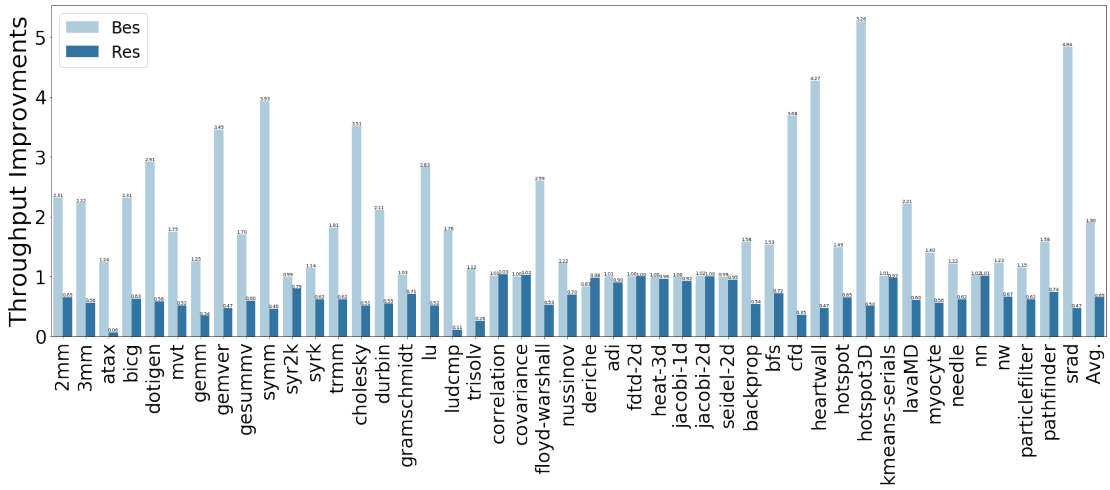


Figure 5.8: Throughput of Beacon Scheduler and Reactive Baseline normalized to CFS on ThunderX2

schedule for these loops hurts the inter-loop data reuse, which is not handled in the beacon scheduler. Trisolv has two reuse loops with large footprints that access the same arrays. While BES saves on memory accesses from thrashing while executing each reuse loop, CFS can save on memory accesses across loops (because CFS does not preempt the application). Hence, we see almost no difference in memory accesses between CFS and BES, but we see differences in performance. Nn has a very small streaming memory footprint of 500B per application (while-loop, expected beacon). This allows ThunderX and ThunderX2 to schedule all processes together as in CFS. However, Hotspot has a mix of both streaming and reuse loops. It fails to create enough memory pressure on ThunderX, but we see an increase in the memory pressure on ThunderX2 and thus an improvement in relative performance.

We present two interesting job completion timelines of Cholesky (which showed substantial benefits with beacons), juxtaposed with those of Correlation (which had no noticeable benefit) (see Figure 5.7). In Cholesky, the beacon scheduler (BES) starts with the same jobs as CFS but soon replaces some of the reuse jobs with other non-cache-pressure types to avoid cache overflow, unlike CFS which thus takes longer to retire its first jobs. BES later on intelligently paces reuse and non-cache-pressure jobs to maintain high throughput, whereas CFS keeps scheduling only the non-cache-pressure types until it finishes the batch. In the case of Correlation, the jobs are within the cache size limit. Thus, BES does not do

anything differently from CFS, and both complete their workloads at roughly the same time.

Reactive Scheduler (RES). The efficiency of the beacon scheduler can be mainly attributed to the prediction of the workload requirements by the compiler inserted beacons. Without the information from the compiler, the scheduler can neither correctly classify the processes, nor allow processes to overlap without hurting one another. We check the utility of this information by replacing this beacon information with online performance counter information. We use the technique used in Merlin [3] to classify the phases as either streaming or reuse. Merlin first uses the cache misses per thousand instructions (MPKI) in LLC to determine memory intensity; then it estimates cache reuse by calculating the memory factor (MF), which is the ratio LLC/(LLC-1) MPKI. A higher MF value indicates less cache reuse because a higher fraction of the misses are served as memory accesses.

We use the same MF threshold as Merlin (0.6) to classify reuse and stream phases. To detect phase change during execution we use instruction per cycle (IPC) degradation as a trigger, similar to Bubble-flux [2] and Merlin. As in Bubble-flux, we use an EPOCH (time-interval) of 250ms to regularly check for state changes during execution. We replace the beacon information by the above information in the beacon scheduler to create a reactive counter-based scheduler called the Reactive Scheduler (RES). The throughput of the BES and the RES normalized with the CFS for the same workloads is shown in Figure 5.8. On average, the beacon scheduler completes the batch 1.9x faster, whereas the RES finishes jobs at a 35% slower rate than the CFS.

5.4 Related Work

Several works leveraged compiler information to predict load for task scheduling [46], reducing power consumption [47], and load-balancing in large clusters [4]. Few works used offline profiling and then augmented that with online data to predict the upcoming phases [38, 39]. [40] targeted throughput scheduling by modeling reuse distance for simple caches by recording the access pattern in a simulator. The reuse-distance information was

then used in predicting the use of L2 cache.

Several works have used performance counters to determine the resource usage of a process. [41] monitors cache usage of threads using reactive counters and tries to co-schedule groups of threads that stay within the cache threshold. [2, 24, 42, 43, 3] all use performance counters to determine the resource sensitivity of a process and the interference of processes. Merlin [3] used the performance counter to determine the streaming versus reuse phase of a process and co-located processes on clusters. Bubble-up, bubble-flux [2, 24] used the reactive counters to generate a QoS versus memory pressure curve offline and then bubble-up used the curve to co-locate processes whereas bubble-flux used the curve to co-locate and the co-schedule lower-priority processes along with high-priority processes. It used IPC degradation to determine the change in resource usage.

Very few of these works have targeted throughput but not as a separate problem of co-scheduling processes, they address the problem of co-location. Bubble-flux co-schedules low-priority processes with high-priority processes to increase machine utilization. BES scheduler co-schedules the same priority processes and maximizes the throughput of the system using the predictive information received from the beacons.

5.5 Conclusions

In this work, we propose Beacon Enabled Scheduler (BES) that proactively co-schedules a large number of jobs to maximize the throughput of the system. The key insight is that the compiler produces predictions, consisting of loop timings and underlying memory footprints along with the type of loop: reuse oriented vs streaming, that are used to make scheduling decisions by the scheduler. A prototype implementation of the framework improves throughput over CFS by up to 4.7x on ThunderX and up to 5.2x on ThunderX2 servers for consolidated workloads.

CHAPTER 6

MINIMIZING LATENCY WITH FAIRNESS THROUGH BEACONS

In the previous chapter, beacons were used to co-schedule the processes targeting the throughput of the machine. While maximizing throughput the latency of the processes and fairness towards each process was ignored. A process could have been de-scheduled for a long time to enable others only for the sole purpose of achieving maximum throughput of the machine. The scheduler only decided on the processes that execute together, but did not arbitrate the exact location (processor, core) on which each of the processes executed. The BES scheduler looked at the Last level cache and the memory bandwidth as the constraints for determining the processes that must be co-scheduled on the machine (single socket). It did not map the internal computing resources to the memory resources, that is the hardware threads, the cores, L1 and L2 caches, sockets and their access to L3 caches and memory were not considered in detail. In other words, BES scheduler focused only on co-scheduling and side-lined the problem of co-location, which arbitrates which processes execute on which core such that the resource conflicts are minimized among the co-executing processes. In this chapter, we propose Bellator, a beacon-enabled latency scheduler that deals with co-location for minimizing the latency of processes while being fair to all processes. Bellator uses the beacon information similar to the throughput scheduler, but instead of deciding on the schedule by arbitrating the simultaneously executing processes, it dynamically co-locates the executing processes on to the cores such that the contention for the resources is minimized thus paving the way to realize minimal latency with complete fairness. Bellator does not time-multiplex the co-executing processes which is left to the underlying CFS.

6.1 Relevant Background

Given the set of applications that are co-scheduled together, co-location deals with what processes are co-located together such that these processes complement resource usage when and where they are determined to be placed. While CFS tries to co-locate the processes such that the compute load is uniformly distributed across the resources, several works have used performance counters to determine the right co-location in different server environments. Many works [50, 51, 52, 24] have tried to find the right co-location by finding the degradation when certain applications or threads are executed together. Bubble-up [24] profiles a fixed set of applications offline to generate QoS versus memory pressure sensitivity curve and then determines what process can be co-located together. Bubble-up does not apply to either a dynamic environment. Bubble-flux [2] uses the memory pressure curve to decide on the co-location of high-priority processes and then uses IPC to co-schedule low-priority processes in a dynamic environment. Bubble-flux improves machine-utilization, however, does not dynamically co-locate the processes. Merlin [3] uses cache-misses and memory-accesses counter to determine the sensitivity towards the cache resource and resource usage to co-locate virtual machines across systems on clusters.

6.2 Bellator

Bellator is designed purely to intake the beacon information and dynamically decide the best place to co-locate the process to finish the upcoming beacon region as shown in Figure 6.1. Bellator does not de-schedule any executing process and aims at achieving overall minimal latency for the processes by minimizing resource conflicts. As introduced earlier CFS is the most prevalent scheduler in the server systems. CFS maintains fairness and distributes the load among the available resources efficiently enabling low latency and high machine utilization for processes in an equal priority environment. Hence, most servers still use CFS for scheduling processes at the system level although a different resource allocation,

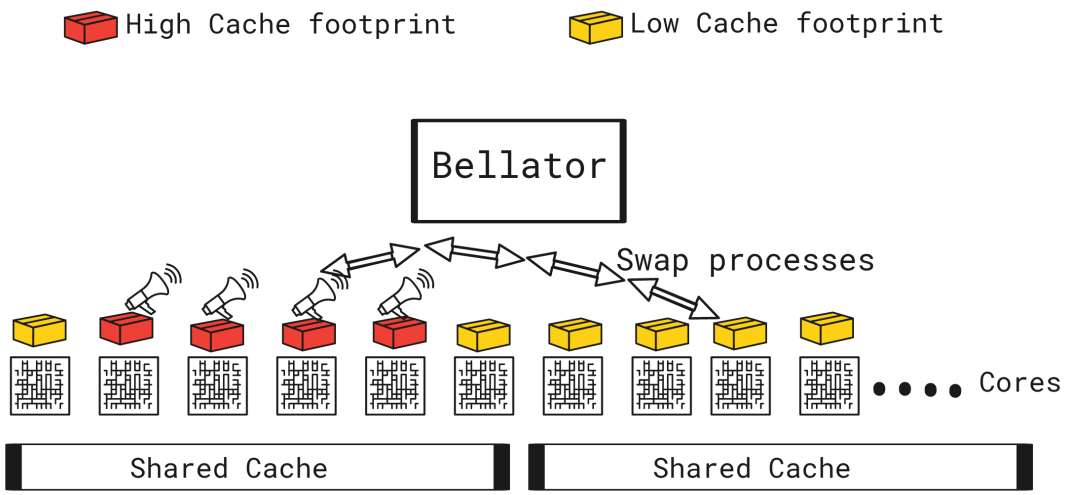


Figure 6.1: Bellator listens to beacons and effectively co-locates processes

load-balancing algorithm may be employed at the cluster level.

Bellator processes the same beacon information available in the previous BES scheduler for throughput, however, utilizes in a manner to co-locate the processes such that the L2 and L3 cache contention and memory bandwidth contention is minimized. The first important difference between the BES scheduler and Bellator is that unlike BES, Bellator does not de-schedule any process and tries to achieve minimal latency for all the executing processes by minimizing the cache conflicts via better dynamic co-location. Bellator is built on top of the principles learnt while developing the BES scheduler and PinIt. Specifically, we confirmed with BES that scheduling streaming processes together even in the face of cache crunch is profitable than scheduling reuse processes together and also that a scheduling conflict can be overlooked when the overlap of the conflicting process region is within 5-10% of the incoming beacon region. Learnt from PinIt, we choose the process that was last scheduled on to a core for arbitrary load-balancing purposes. This ensures that we select the process with the least warmed private cache. These principles can be seen throughout the algorithm of the Bellator which decisively places the incoming beacon regions to avoid cache and memory bandwidth conflicts.

Bellator models sockets, cores, hardware threads, and their respective caches – L3,

L2, and L1 – according to their physical relation. For example, Bellator models the exact hardware config of Thunderx2, which has two sockets with each consisting of 28 cores and an L3 cache and each core entailing four hardware threads that share the L1 and L2 cache. Bellator's scheduling is at the granularity of the core, that is, a process is co-located from one core to another and the underlying CFS takes care of scheduling to the hardware thread. Also, Bellator looks for reducing conflict in L2 and L3 cache. The arbitration for the L2 cache also takes care of the L1 cache, however, the L1 cache can be spilled. Finer the granularity of arbitration more logic and more the overhead. Hence, Bellator compromises granularity to reduce overhead. Bellator relies on beacons for the process's resource requirement information similar to BES. Bellator wakes to perform co-location on four events–process launch, beacon start, beacon stop, and process exit. Bellator parses all the job commands and launches each of the applications. On every launch, Bellator fairly distributes the process among the sockets and the cores. Every resource is filled before a previously allocated resource is re-allocated, that is, every process is assigned to a new domain before re-assigning to the previously seen domain. The domain here refers to either socket or core. Bellator currently does not create domains of pairs of cores or sockets with the same hops as in scheduling domains of the CFS. This provides more flexibility for resource assignment. However, as future work, the domains in Bellator can be matched to that of the CFS to see if the granularity reduces scheduling logic overhead. Note that although Bellator is a scheduler and the algorithm is referred to as scheduling logic, Bellator only determines the best co-location dynamically for the incoming beacon processes and does not interfere with time-slicing or even scheduling at hardware thread level. This is left to the underlying CFS.

The socket and core for the new process that has to be launched is determined by the method shown in Algorithm 5. According to the launching process number, Bellator picks the socket to assign based on the number of sockets available (Line 1 -2) in Algorithm 5. After selecting the socket, based on the number of processes already executing on the socket,

Algorithm 5 Bellator Process Launch

```
1: procedure PROCESS_LAUNCH(process_num)
2:     num_sockets ← #sockets
3:     s_i ← socket mod (process_num, num_sockets)
4:     num_socket_procs = #procs in s_i
5:     num_cores ← #cores in s_i
6:     c_i ← core mod (num_socket_procs, num_cores) ∈ s_i
7:     assign process process_num to c_i ∈ s_i
```

Bellator sequentially assigns the next core to the process as in Line 4-7. The executing process is assigned to a core by using **taskset** in Linux, which pins the process to the hardware threads of the core. The process starts executing and then sends out beacons which are handled by Bellator's **Handle_Start_Beacon** consisting of multiple subprocedures all shown from Algorithm 6 to Algorithm 13.

On seeing a start beacon, the beacon before the start of the loop, Bellator invokes the Procedure **Handle_Start_Beacon** shown in Algorithm 11. The procedure first stores the result of whether the current socket can satisfy the cache requirements of the process (Line 94) by calling **Can_Socket_Satisfy** procedure outlined in Algorithm 6. This call checks if the L3 cache has enough space to house the memory footprint of the process. The cache requirement (memory footprint) of each process are classified into buckets to index processes with similar requirements. A bucket for a given memory footprint is the base two logarithmic value of the footprint starting from an initial footprint value, defined by $INIT\_CACHE\_BUCKET$, as shown in the procedure **Get_Cache_Bucket** in Algorithm 6 Line 5. For example, the initial bucket is 512 bytes, which implies footprint values less than or equal to 512 bytes are assigned to bucket 0. Also, the values ranging from 512 to 1023 (one previous to the next power of 2 which is 1024) will also be assigned to bucket 0. However, values from 1024 to 2095 will be assigned to bucket 1, and so on. The processes within the same bucket are treated similarly and are replaced with one another in the co-location algorithm. This allows for easy access to the different processes and classifies the requirements well within each level of the caches which change from KB to

MBs in size. Back to **Can_Socket_Satisfy**, if enough cache is not available, the processes within the same cache bucket are checked to determine the overlap with the incoming process. If the overlap is within a 5-10% (configurable) as in the BES scheduler, then the socket can satisfy the requirement.

Algorithm 6 Bellator Handle Start Beacon

```
1:  procedure GET_CACHE_BUCKET(Size sz)
2:      if sz == 0 then
3:          return 0
4:      else
5:          return log2(sz / INIT_CACHE_BUCKET)
6:  procedure CAN_SOCKET_SATISFY(Socket s, Process p)
7:      cr ← Cache Requirement of Process p
8:      ca ← L3 Cache Available in s
9:      if ca >= cr then
10:         return True
11:     else
12:         cq = abs(ca − cr)
13:         buc ← GET_CACHE_BUCKET(cq)
14:         P ← {j1, ..., jm} ∀ ji ∈ s ∈ buc
15:         for j ∈ P do
16:             if OVERLAP(p, j) < Threshold_Overlap then
17:                 return True
18:     return False
```

Secondly in **Handle_Start_Beacon**, the result of whether the current socket can stream the process requirements is checked by calling **Can_Socket_Stream** (Line 95). In the procedure outlined in Algorithm 7, if the process is of **REUSE** type, then there is no question of streaming the process memory, if not, then similar to **Can_Socket_Satisfy**, the procedure checks if there is enough bandwidth within the socket, or else if a process within the same bandwidth bucket has a timing overlap within the threshold, for the socket to stream the incoming process.

Next in **Handle_Start_Beacon**, based on the memory footprint, the process is handled differently. if the requirement is less than L2, or greater than L2 and less than L3, or greater than L3. This helps Bellator discern the right schedule within the socket, across sockets,

Algorithm 7 Bellator Handle Start Beacon

19: **procedure** GET_BW_BUCKET(Bandwidth bw)
20: **if** $bw == 0$ **then**
21: **return** 0
22: **else**
23: **return** $\log_2(\frac{bw}{INIT\_BW\_BUCKET})$
24: **procedure** CAN_SOCKET_STREAM(Socket s, Process p)
25: **if** Beacon Type$(p) == REUSE$ **then**
26: **return** $True$
27: $bw \leftarrow$ memory bandwidth of process p
28: $ba \leftarrow$ memory bandwidth available in s
29: **if** $ba >= bw$ **then**
30: **return** $True$
31: **else**
32: $bq = abs(ba - bw)$
33: $buc \leftarrow$ GET_BW_BUCKET(cq)
34: $P \leftarrow \{j_1, ..., j_m\} \forall j_i \in s \in buc$
35: **for** $j \in P$ **do**
36: **if** OVERLAP$(p, j) < Threshold_Overlap$ **then**
37: **return** $True$
38: **return** $False$

and process that do not fit within the socket. If the process cache requirement is less than L2, the process must be scheduled onto a core that has enough space on L2. However, the additional cache demand must also be fulfilled within L3. If the current socket cannot satisfy, then if any other socket can satisfy the cache requirement and also stream the process, then the socket is marked as the destination socket (Line 97-101). If no other socket can satisfy the process requirements the current socket is still selected to satisfy as the cache requirements are still within the L2 size (Line 103). If the current socket is selected, then Bellator first checks whether the current core can satisfy the L2 cache requirements by calling **Can_Core_Satisfy** (Line 104). The procedure outlined in Algorithm 8, first checks whether L2 has enough space, if not then checks among the beacon processes executing on the core for a process which on finishing the beacon region releases enough space for the incoming process and the overlap between the beacon-regions is within the Threshold overlap (Line 46-49). If the procedure returns false, then Bellator checks if any other core in

the socket can satisfy the process requirement (Line 107-110).

Algorithm 8 Bellator Handle Start Beacon

```
39: procedure CAN_CORE_SATISFY(Core c, Process p)
40:     cr ← Cache Requirement of Process p
41:     ca ← L2 Cache Available in c
42:     if ca >= cr then
43:         return True
44:     else
45:         cq = abs(ca − cr)
46:         for Beacon Process p_i ∈ c do
47:             cr_i ← Cache Requirement of Process p_i
48:             if cr_i + ca <= cr  &  OVERLAP(p, p_i) < Threshold_Overlap then
49:                 return True
50:     return False
```

If no core can satisfy the cache requirement, Bellator checks by calling **Can_AnyCore_Barter_With**, if any core can exchange a process such that both processes cache requirements are met. Inside the call, the procedure, shown in Algorithm 9, checks only among the cores that have the available cache space equal to one below the actual bucket of the process's cache requirement (Line 54). This avoids checking processes from all the cores, instead. The intuition is that a core that already has enough cache for satisfying one bucket lower cache requirement might have a process that can be bartered. For each such core, Bellator checks if any beacon process can be bartered with the current core in Line 61 and 62. The procedure returns a process that can be bartered with the current process. If in **Handle_Start_Beacon**, a different socket was selected than the current socket, the cores in the destination socket are checked to find a core that can satisfy the cache requirements of the process (Line 114 -117). If not, Bellator calls **Can_AnyCore_Barter_With** to check any core/process in the destination socket can barter with the current socket and current core (Line 119). If a destination core is found, then the process is moved to the new core by calling **Change_Core**, shown in Algorithm 11. Instead, if a process to barter with is found, then **Barter_Procs** is called which swaps the two processes among the cores executing them as shown in Algorithm 10.

Algorithm 9 Bellator Handle Start Beacon

```
51: procedure CAN_ANYCORE_BARTER_WITH(Socket s, Core c, Process p)
52:     cr ← Cache Requirement of Process p
53:     buc ← GET_CACHE_BUCKET(cr) − 1
54:     C ← {c_1, ..., c_m} ∀c_i ∈ s  &
          GET_CACHE_BUCKET(Available cache in c_i) == buc
55:     ca ← L2 Cache Available in c
56:     for c_i ∈ C do
57:         for Beacon process p_i ∈ C do
58:             cr_i ← Cache Requirement of Process p_i
59:             ca_i ← L2 Cache Available in c_i
60:             if cr_i < ca  &  ca_i + cr_i <= cr then
61:                 return p_i
62:     return NULL
```

If no such destination core or barter process is found, then Bellator checks if within the destination socket or the current socket there exists a core with more space than the current core by calling **Find_Better_core_If_Any** on Line 127 in Algorithm 12. Specifically, the procedure **Find_Better_core_If_Any**, outlined in Algorithm 10, starting with the bucket equal to that of L2 and till the bucket next to the current core's available cache bucket or 0 in case no current core is specified, checks if there are cores with available caches equal to these buckets. If the process' beacon type is not of **REUSE** type, then the first core of the highest bucket is returned. Else, among the cores from the highest bucket, the core with the least number of reuse processes is returned. The logic is to distribute the reuse processes such that the reuse processes do not execute together if it can be helped to be executed with a stream or non-beacon process. If the procedure returns a different core, then Bellator first checks if a process exists on the returned core that can fit on the current core on Line 129. If so the processes are bartered or else the current process is moved to the returned core (Line 130-133). Since a proper fit for the process was not found, it is marked as deferred for further re-scheduling when a stop beacon event occurs, explained later. It is evident that bartering processes is more preferred in general than simply moving the process in question to a different core because bartering maintains the invariant of fair distribution that was initially formed during the process launch.

Algorithm 10 Bellator Handle Start Beacon

63: **procedure** FIND_BETTER_CORE_IF_ANY(socket s,core c, process p)
64: **if** $c == NULL$ **then**
65: $lowest\_Bucket = 0$
66: **else**
67: $lowest\_Bucket =$ GET_CACHE_BUCKET(Available cache in c) $+ 1$
68: $l2\_Bucket =$ GET_CACHE_BUCKET($L2\_size$)
69: $buc = l2\_Bucket$
70: **while** $buc > lowest_Bucket$ **do**
71: $C \leftarrow \{c_1, ..., c_m\} \forall c_i \in s$ & GET_CACHE_BUCKET(Available cache in c_i) $==$ buc
72: **if** $|C| > 0$ **then**
73: **if** Beacon Type (p) $\neq REUSE$ **then**
74: **return** $c_1 \in C$
75: **else**
76: **return** $c_k \in Cs.t.c_k = \min_{c_i \in C}$(#reuse procs $\in c_i$)
77: **return** $NULL$
78: **procedure** BARTER_PROCS(process p_1, process p_2)
79: $c_1 \leftarrow$ core executing p_1
80: $c_2 \leftarrow$ core executing p_2
81: remove process p_1 from c_1
82: remove process p_2 from c_2
83: insert process p_1 in c_2
84: insert process p_2 in c_1

Next in **Handle_Start_Beacon** the process with cache requirement greater than L2 but less than L3 is handled. If the current socket cannot satisfy the process cache requirement, then Bellator checks other sockets if any can stream and satisfy the cache requirement (Line 137 -142). While checking if any other socket can satisfy, Bellator also tracks the socket with most available cache and the core in the socket with maximum available cache (Line 147-150). If no socket could satisfy the process requirement completely, then Bellator knows which socket and core can satisfy the requirements as much as possible. If the current socket itself can satisfy and stream or if the current socket is the one with maximum cache, then Bellator just finds the better core within the socket (Line 153). If a barter process exists in the destination core, then Bellator swaps the processes or else just assigns the destination core to the incoming process (Line 157-162). In case no socket could satisfy or stream the process requirements completely, the process is marked as deferred, but still moved to a core with more cache (Line 154 -156).

In case the process's cache requirement is greater than L3, all Bellator can do is fairly distribute the processes that are similar such that no resource (i.e. socket) is over-burdened. Bellator first gets the number of processes in the current socket that belong to the same bucket (Line 164 -166 in Algorithm 13). If more than one such processes exist in the current socket, Bellator checks all other sockets to find a socket that has at least two less such processes (Line 167 - 173). Bellator continues to search the sockets to find one with least such processes. Once a socket is found, Bellator searches for processes in lower buckets to barter with the current processes for fair distribution (Line 184-188). However, if no socket has the number of processes in the same bucket fewer by two, then if the current processes is a reuse process, then Bellator checks for the number of stream processes in the same bucket. The idea is to distribute the stream processes equally so that co-execution of reuse processes is reduced. If stream processes are fewer by two than in any other socket, then a stream process from the other socket is used to barter with the current reuse process (Line 174-181). Once a barter process is swapped to the current socket, Bellator calls

Algorithm 11 Bellator Handle Start Beacon

```
85: procedure CHANGE_CORE(process p, core c)
86:     c_c ← core executing p
87:     remove process p from c_c
88:     insert process p in c
89: procedure HANDLE_START_BEACON(process p)
90:     c_c ← core executing p
91:     s_c ← socket s.t. c_c ∈ s_c
92:     cr ← memory footprint of process p
93:     bw ← memory bandwidth of process p
94:     can_soc_satisfy_cr ← CAN_SOCKET_SATISFY(s_c, p)
95:     can_soc_stream ← CAN_SOCKET_STREAM(s_c, p)
96:     if cr < L2size then
97:         if can_soc_satisfy == False || can_soc_stream == False then
98:             for all Socket s_i ≠ s_c do
99:                 if CAN_SOCKET_STREAM(s_i, bw)
                     & CAN_SOCKET_SATISFY(s_i, cr) then
100:                    destination socket s_d ← s_i
101:                    break
102:        if s_d == NULL then
103:            s_d ← s_c
104:            if CAN_CORE_SATISFY(c_c, cr) then
105:                destination core c_d ← c_c
106:            else
107:                for all Core c_i ∈ s_c ≠ c_c do
108:                    if CAN_CORE_SATISFY(c_i, cr) then
109:                        c_d ← c_i
110:                        break
111:            if c_d == NULL then
112:                Barter proc p_b ← CAN_ANYCORE_BARTER_WITH(s_c, c_c, p)
113:        else
114:            for all Core c_j ∈ s_d do
115:                if CAN_CORE_SATISFY(c_j, cr) then
116:                    c_d ← c_j
117:                    break
118:            if c_d == NULL then
119:                p_b ← CAN_ANYCORE_BARTER_WITH(s_d, c_c, p)
120:        if c_d ≠ NULL then
121:            if c_d ≠ c_c then
122:                CHANGE_CORE(p, c_d)
123:        else
124:            if p_b ≠ NULL then
125:                BARTER_PROCS(p, p_b)
126:            else
```

Algorithm 12 Bellator Handle Start Beacon

```
                    c_d ← FIND_BETTER_CORE_IF_ANY(s_e, p, c_c) s.t. s_e ← s_d or s_c
                    if c_d ≠ NULL & c_d ≠ c_c then
                        p_b ← process that can fit in c_c
                        if p_b ≠ NULL then
                            BARTER_PROCS(p, p_b)
                        else
                            CHANGE_CORE(p, c_d)
                    deferred ← True
    else
        if cr < L3size then
            if can_soc_satisfy_cr == False then
                max_available_cache ← available cache in s_c
                for all Socket s_i ≠ s_c do
                    if CAN_SOCKET_STREAM(s_i, bw) then
                        continue
                    if CAN_SOCKET_SATISFY(s_i, cr) then
                        destination socket s_d ← s_i
                        c_d ← FIND_BETTER_CORE_IF_ANY(s_i, p, NULL)
                        break
                    else
                        if available cache in s_i > max_available_cache then
                            s_d ← s_i
                            c_d ← FIND_BETTER_CORE_IF_ANY(s_i, p, NULL)
                            max_available_cache ← available cache in s_i
            else
                s_d ← s_i
                c_d ← FIND_BETTER_CORE_IF_ANY(s_c, p, c_c)
            if s_d == NULL then
                deferred ← True
                c_d ← FIND_BETTER_CORE_IF_ANY(s_c, p, c_c)
            if c_d ≠ NULL & c_d ≠ c_c then
                p_b ← process that can fit in c_c
                if p_b ≠ NULL then
                    BARTER_PROCS(p, p_b)
                else
                    CHANGE_CORE(p, c_d)
        else
            buc ← GET_CACHE_BUCKET(cr)
            P ← {j_1, ..., j_m} ∀ j_i ∈ s_c ∈ buc
            m = |P| + 1
```

Algorithm 13 Bellator Handle Start Beacon

if $m > 1$ **then**
  for all Socket $s_i \neq s_c$ **do**
    $Q \leftarrow \{k_1, ..., k_r\} \forall k_i \in s_i \in buc$
    $r = |Q|$
    if $r < (m - 1)$ **then**
      $m = r$
      Barter socket $s_b \leftarrow s_i$
  if $s_b == NULL$ & Beacon Type$(p) == REUSE$ **then**
    $m_s \leftarrow |\{j_b, ...j_l\}| \forall j_i \in P$ & Beacon Type$(j_i) == STREAM$
    for all Socket $s_i \neq s_c$ **do**
      $Q \leftarrow \{k_1, ..., k_r\} \forall k_i \in s_i \in buc$
      $r_s \leftarrow |\{k_b, ...k_l\}| \forall (k_i \in Q)$
        & Beacon Type$(k_i) == STREAM$
      if $r_s > m_s$ **then**
        $m_s = r_s$
        $s_b \leftarrow s_i$
  if $s_b \neq NULL$ **then**
    $i = buc - 1$
    while $i >= 0$ & $p_b == NULL$ **do**
      if $|\{k_1, ..., k_r\}| > 0 \forall k_i \in s_b \in buc$ **then**
        $p_b \leftarrow k_1$
        break
      $i = i - 1$
    if $p_b \neq NULL$ **then**
      BARTER_PROCS(p, p_b)
      $c_p \leftarrow$ FIND_BETTER_CORE_IF_ANY(s_c, p_b, c_c)
      if $c_p \neq NULL$ & $c_p \neq c_c$ **then**
        $p_v \leftarrow$ process that can fit in c_c
        if $p_v \neq NULL$ **then**
          BARTER_PROCS(p_b, p_v)
        else
          CHANGE_CORE(p_b, c_p)
  else
    $c_d \leftarrow$ FIND_BETTER_CORE_IF_ANY(s_c, p, c_c)
    if $c_d \neq NULL$ & $c_d \neq c_c$ **then**
      $p_b \leftarrow$ process that can fit in c_c
      if $p_b \neq NULL$ **then**
        BARTER_PROCS(p, p_b)
      else
        CHANGE_CORE(p, c_d)
  $deferred \leftarrow True$
assign $cr, bw$to process$p$

Find_Better_Core_If_Any to check and find if another core has more L2 cache available than the current core and then either barters or changes core accordingly for the bartered proc (Line 191-197). If no process is found either from lower buckets or from streaming pool for bartering, but the distribution is still unfair with another socket, then the current process is moved to the other socket and a better core is found, if any, within the socket.

If the distribution of similar processes along with stream processes is already fair, then Bellator checks if any other core has more cache than the current cache by calling **Find_Better_Core_If_Any** and barters the processes if it can barter with the better core or just moves the process to the core. All the processes that are greater than L3 are deferred, so that these processes are fairly distributed whenever there is a change in the distribution of such a process.

For any incoming process, once the destination core and socket for the process is finalized, the resources required by the process is assigned from the destination core and socket (Line 207). Note that, the current socket and core can also be the destination socket and core, respectively, if they can satisfy the process requirements.

The process after sending a start beacon does not wait for the scheduler and continues executing the beacon region. Soon after executing the region, the process sends a stop beacon to the scheduler. The scheduler on seeing a stop beacon calls **Handle_Stop_Beacon** outlined in Algorithm 14 and 15. The Algorithm in Bellator for handling stop beacon first reclaims the allocated resources on Line 6. If the current process was set as deferred, then it is reset back as not deferred. If this procedure was called to handle missed stop beacon, that is another start beacon for the same process was seen before seeing a stop beacon, then the deferred processes are not re-scheduled and the procedure returns to handle the new start beacon. The idea is that a beacon for the current process is already waiting and will probably consume the current core and socket within which the process was executing. Hence using this core, socket to re-schedule deferred processes can result in unnecessary movement of the current process and its caches.

If a beacon was not missed, then Bellator tries to use the current core and socket to re-schedule a deferred process. Starting with the deferred processes that belong to the cache bucket to which the released cache belonged till the last bucket (0^{th} bucket), are checked and re-scheduled as long as at least one such process is marked as not deferred. For a deferred process in the bucket within the loop at Line 12, Bellator first checks and stores whether the socket can satisfy the process cache requirement and whether the socket can stream the process by calling **Can_Socket_Satisfy** and **Can_Socket_Stream**, respectively, similar to the **Handle_Start_Beacon** procedure. Also, similarly to the start beacon, the process is handled differently based on whether the cache requirement of the deferred process is within L2, or greater than L2 and within L3, or greater than L3.

If the cache requirement of the deferred process is lower than L2, then if the current socket can satisfy the requirement in the L3 cache and also stream the process and if the current core can satisfy the cache requirement, the process is moved to the current core and the process is unmarked as deferred and once the core was used there is no other free core yet for other deferred processes so Bellator returns from the **Handle_Stop_Beacon** procedure (Line 17-22). If the cache requirement is between L2 and L3 and if the socket can satisfy and stream the process requirements, then Bellator checks if any other core has more available L2 space than the current core by calling **Find_Better_Core_If_Any**. If a process on either the current core or the better core can fit on the deferred process core, then the processes are swapped or else the deferred process is moved to either current or better core. The process is unmarked from deferred.

If the deferred process's cache requirement exceeds L3 and belongs to a different socket, then Bellator first checks if the number of similar processes that belong to this bucket are greater by at least two in the other socket compared to the current socket. If so, then the processes must be moved to the current socket. Bellator finds a better core if any (Line 47), and exchanges the deferred process with a process that can fit the core executing the deferred process. If a barter process is not found, then the deferred process is simply moved

to the better core or current core. If sockets have the same number of processes belonging to this bucket, then if the deferred process is of reuse type, Bellator checks for the difference in the number of stream processes in the deferred process's socket and the current socket. If the current socket has more streaming processes in the same bucket than the other socket, one of the streaming processes is bartered for the deferred reuse process (Line 55-59). The idea is to fairly distribute the large processes among the sockets and also fairly distribute the stream processes. The deferred process is not unmarked because these large processes are never satisfied and are always re-distributed to keep the load fair among the sockets. Bellator moves on to the next bucket because this bucket processes are now re-distributed fairly if it was not already.

Once a process finishes executing, Bellator is notified and **Process_Exit** is called. The procedure first removes the process from the core and socket that it was last executing on and then carries out redistribution by calling **Redistribute_On_Exit** as shown in Algorithm 16. To redistribute, the procedure checks if any other socket has more processes than the current process. If so, then a non-beacon process from the other socket is moved to the current core in the socket because a non-beacon process does not need algorithmic co-location. If sockets have equal load or if a non-beacon process does not exist to balance the processes on the sockets and the current core has no processes, then a process from a core that has more than one process is moved to the current core.

Fairness. Similar to CFS, Bellator schedules fairly by ensuring that no process is starving and also balancing the load among the computing resources. In addition, Bellator also ensures that the cache and memory bus pressure is fairly distributed among the resources by attempting to fit the cache requirements and distributing the stream and reuse processes among the available cores and sockets. If anything, this distribution adds to the fairness of the dynamic environment in which every process is executing.

Algorithm 14 Bellator Handle Stop Beacon

```
1: procedure HANDLE_STOP_BEACON(process_p)
2:     c_c ← core executing p
3:     s_c ← socket s.t. c_c ∈ s_c
4:     cr ← memory footprint of process p
5:     bw ← memory bandwidth of process p
6:     release cr, bw ∈ c_c ∈ s_c
7:     if missedbeacon then
8:         return
9:     buc ← GET_CACHE_BUCKET(cr)
10:    while buc >= 0 do
11:        P ← {j_1, ..., j_m} ∀ j_i ∈ buc & deferred == True
12:        for all p_i ∈ P do
13:            soc_can_satisfy ← CAN_SOCKET_SATISFY(s_c, p_i)
14:            can_soc_stream ← CAN_SOCKET_STREAM(s_c, p_i)
15:            cq ← memory footprint of process p_i
16:            if cq < L2_size then
17:                if soc_can_satisfy == True & can_soc_stream == True
                       & CAN_CORE_SATISFY(s_c, c_c, p_i) == True then
18:                    c_i ← core executing p_i
19:                    if c_i ≠ c_c then
20:                        CHANGE_CORE(p_i, c_c)
21:                    deferred ← False
22:                    return
23:            else
24:                if cq < L3_size then
25:                    if soc_can == True & can_soc_stream == True then
26:                        c_p ← c_c
27:                        c_p ← FIND_BETTER_CORE_IF_ANY(s_c, p_i, c_c)
28:                        c_i ← core executing p_i
29:                        if c_p ≠ NULL & c_p ≠ c_i then
30:                            p_v ← process ∈ c_p that can fit in c_i
31:                            if p_v ≠ NULL then
32:                                BARTER_PROCS(p_i, p_v)
33:                            else
34:                                CHANGE_CORE(p_i, c_p)
35:                        deferred ← False
36:                        return
37:                else
```

Algorithm 15 Bellator Handle Stop Beacon

```
38:                 c_i ← core executing p_i
39:                 s_i ← socket s.t. c_i ∈ s_i
40:                 if s_i ≠ s_c then
41:                     P ← {j_1, ..., j_m}∀j_i ∈ s_c ∈ buc
42:                     Q ← {k_1, ..., k_r}∀k_i ∈ s_i ∈ buc
43:                     m = |P|
44:                     r = |Q|
45:                     if m < (r − 1) then
46:                         c_d ← c_c
47:                         c_d ← FIND_BETTER_CORE_IF_ANY(s_c, p_i, c_c)
48:                         if c_d ≠ NULL & c_d ≠ c_i then
49:                             p_v ← process ∈ c_d that can fit in c_i
50:                             if p_v ≠ NULL then
51:                                 BARTER_PROCS(p_i, p_v)
52:                             else
53:                                 CHANGE_CORE(p_i, c_d)
54:                     else
55:                         if BeaconType(p_i) == REUSE then
56:                             P_s ← {j_a, ....j_h}∀(j_i ∈ P)
                                    & Beacon Type(j_i) == STREAM
57:                             Q_s ← {k_b, ....k_l}∀(k_i ∈ Q)
                                    & Beacon Type(k_i) == STREAM
58:                             if |P_s| > |Q_s| then
59:                                 BARTER_PROCS(p_i, j_h)
```

Algorithm 16 Bellator Process Exit

```
1:  procedure REDISTRIBUTE_ON_EXIT(socket_p, core_p)
2:      for all Socket s ≠ socket_p do
3:          if #procs ∈ socket_p < (#procs ∈ s) − 1 then
4:              p_b ← non-beacon proc ∈ s
5:              if ∃p_b then
6:                  move p_b to core_p ∈ socket_p
7:                  redistributed ← True
8:      if redistributed == False then
9:          if #procs ∈ core_p == 0 then
10:             for all non-beacon-proc p_i ∈ socket_p do
11:                 c_i ← core executing p_i
12:                 if #procs ∈ c_i > 1 then
13:                     remove process p_i from c_i
14:                     insert process p_i in core_p
15:                     break
16: procedure PROCESS_EXIT(process_p)
17:     s_p ← socket executing process_p
18:     c_p ← core ∈ s_i executing process_p
19:     remove process_p from core c_i
20:     remove process_p from socket s_i
21:     REDISTRIBUTE_ON_EXIT(s_p, c_p)
```

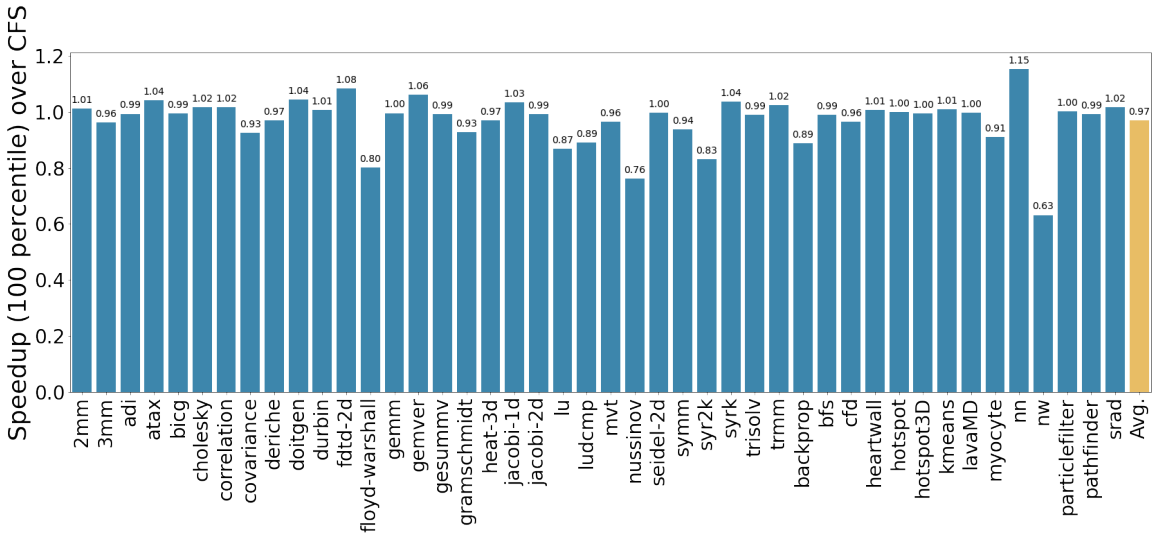


Figure 6.2: Speedup (100 th percentile) for 27 processes job configuration

6.3 Experiments

We conducted our experiments on ThunderX2. All our experiments used only 27 cores on each socket by using the taskset command for CFS and through appropriate configuration files for Bellator. Similar to BES, Bellator is a derived class over the base scheduler class, which performs CFS scheduling by leveraging the underlying Linux, implemented in C++. The difference in Bellator is the usage of the information and its consequent outcomes. The total computing entities which Linux determines as a CPU is 224, with 122 belonging to each socket. We do not use one core each in the two sockets and leave it free for other daemon processes. Hence, we see the total computing units as 216 with 118 in each socket. We stress the scheduler at mainly six different simultaneous processes configuration – 27, 54, 108, 162, and 216 simultaneous applications. We used CFS in the Linux kernel version 4.15 as the baseline scheduler.

We conduct our experiments using Polybench and Rodinia as used in BES experiments. We train on one set of inputs of different size ranges and then change the values of all these inputs because the job configuration is made up of a mixture of all the different input sizes. The six different configurations mentioned above are made of only mini, small, medium,

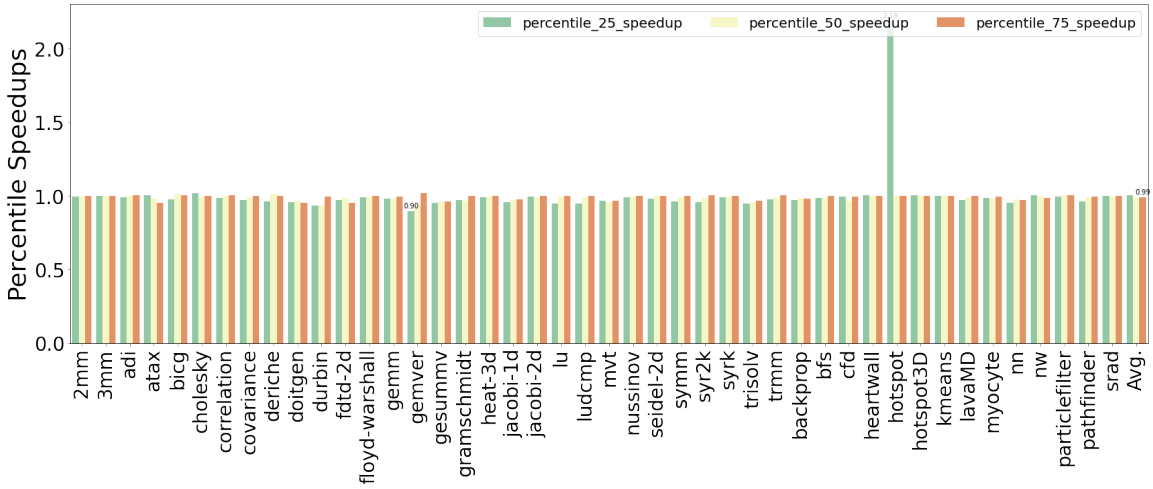


Figure 6.3: Speedup at 25, 50, 75 th percentile of processes completion for 27 processes job configuration

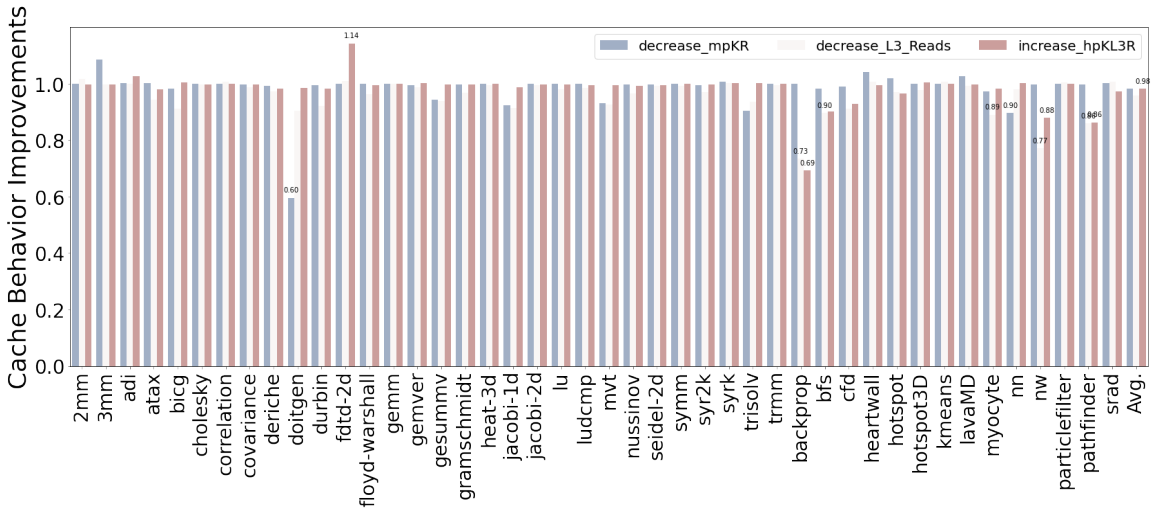


Figure 6.4: Cache Behavior in terms of cache misses, L3 reads and hits for 27 processes job configuration

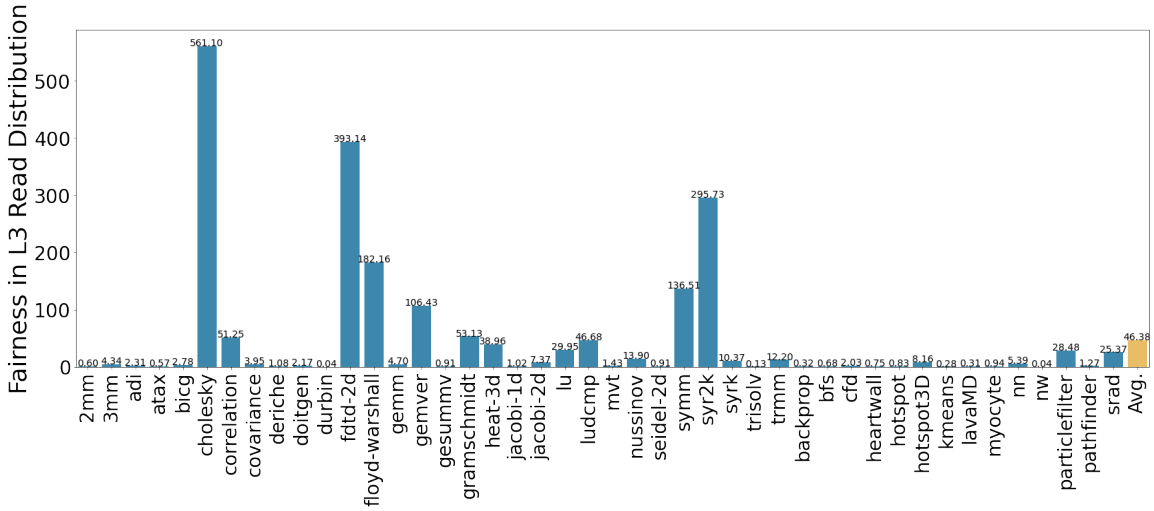


Figure 6.5: Fairness in load distribution among L3 caches for 27 processes job configuration

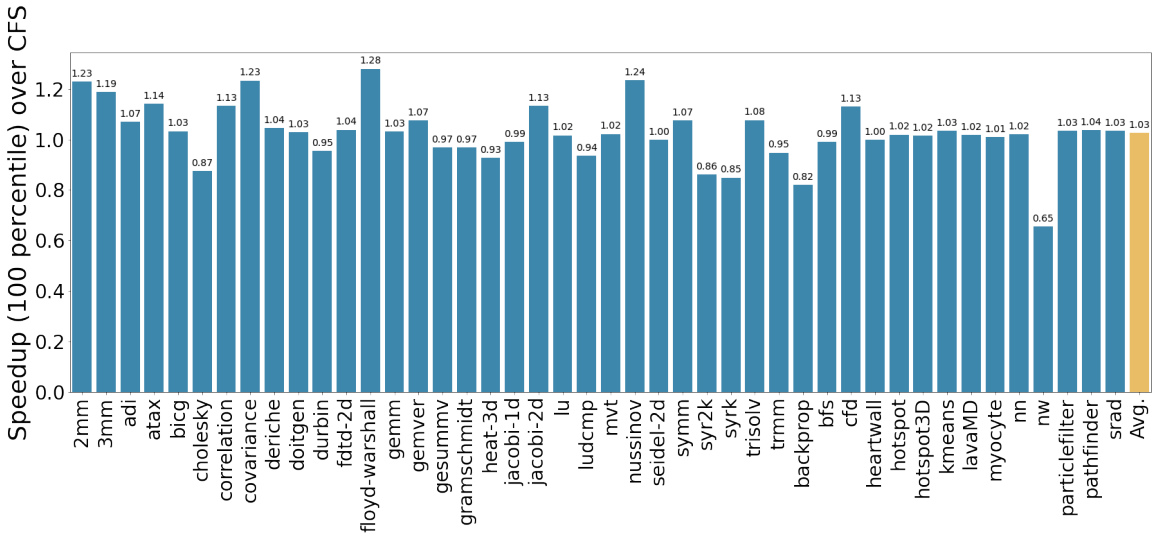


Figure 6.6: Speedup (100 th percentile) for 54 processes job configuration

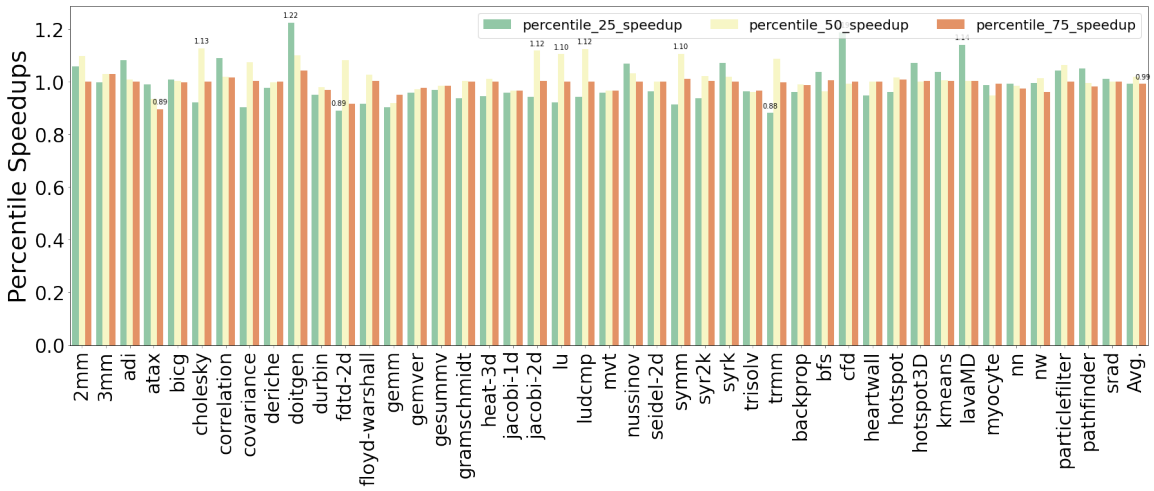


Figure 6.7: Speedup at 25, 50, 75 th percentile of processes completion for 54 processes job configuration

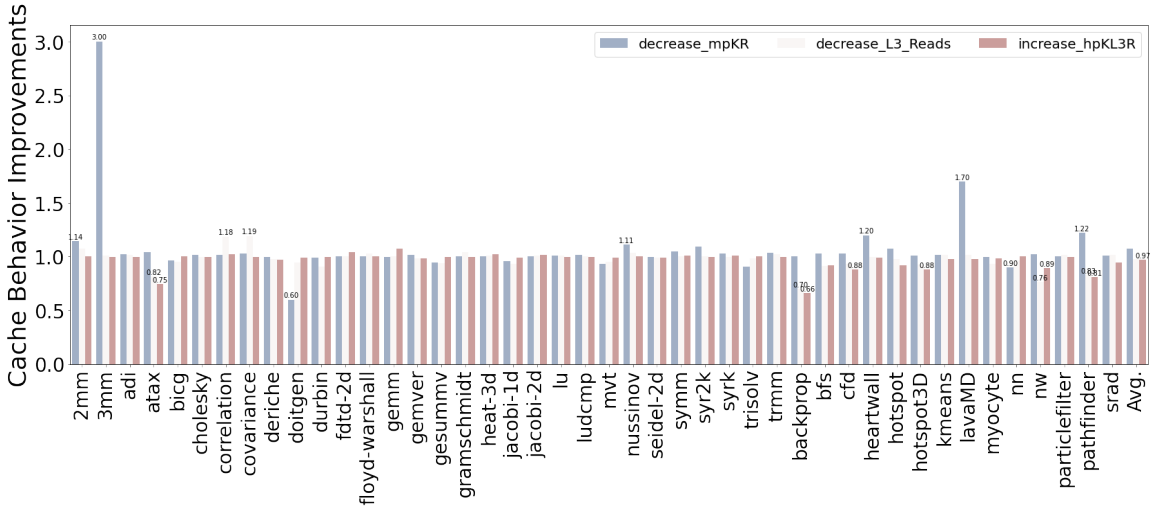


Figure 6.8: Cache Behavior in terms of cache misses, L3 reads and hits for 54 processes job configuration

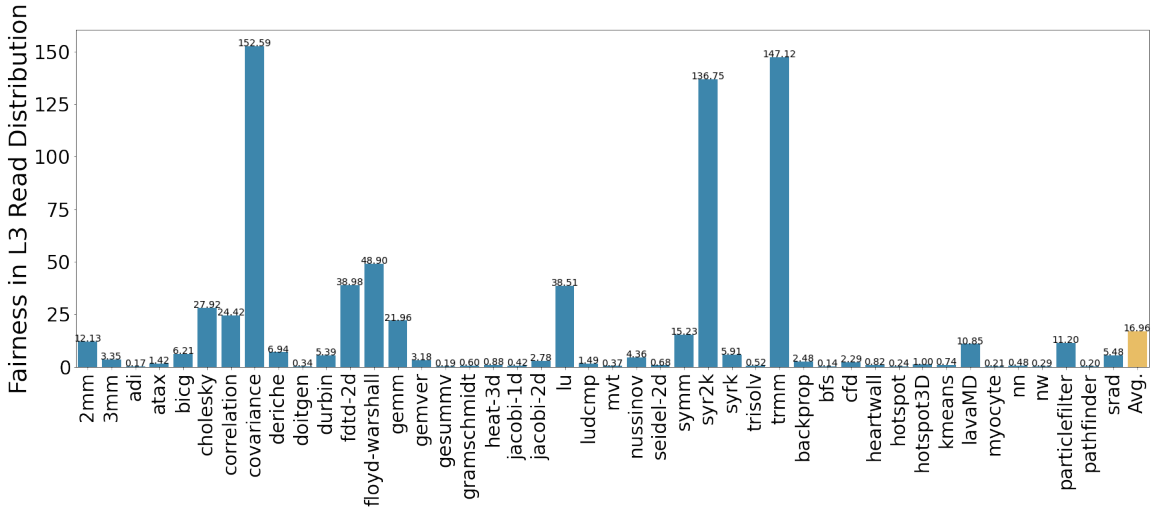


Figure 6.9: Fairness in load distribution among L3 caches for 54 processes job configuration

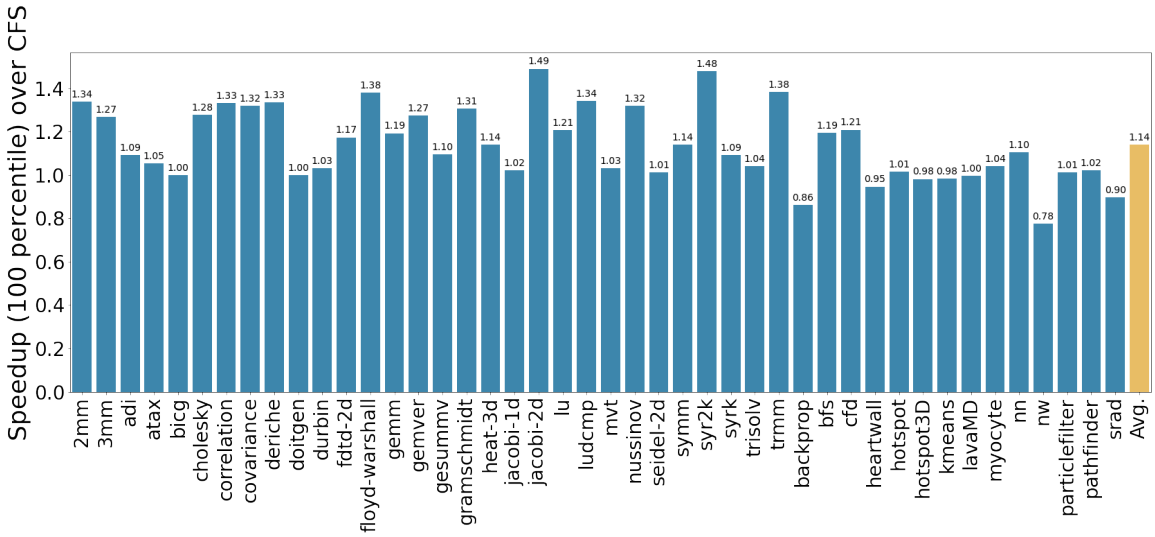


Figure 6.10: Speedup (100 th percentile) for 108 processes job configuration

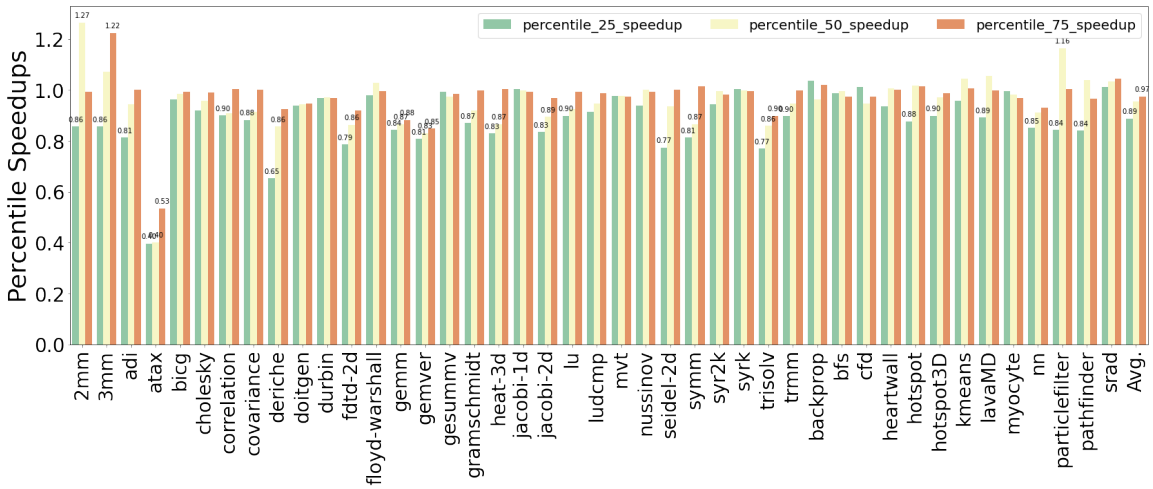


Figure 6.11: Speedup at 25, 50, 75 th percentile of processes completion for 108 processes job configuration

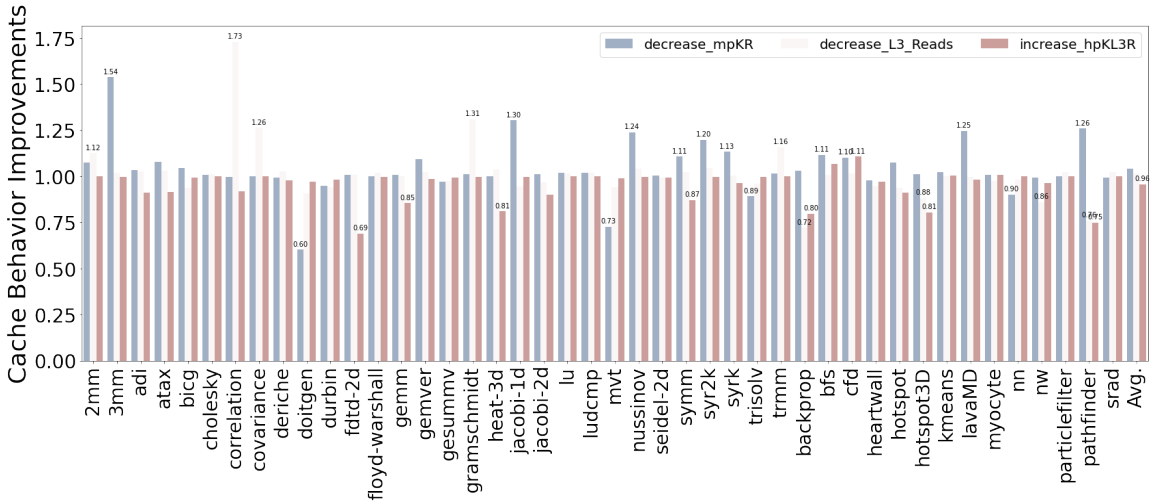


Figure 6.12: Cache Behavior in terms of cache misses, L3 reads and hits for 108 processes job configuration

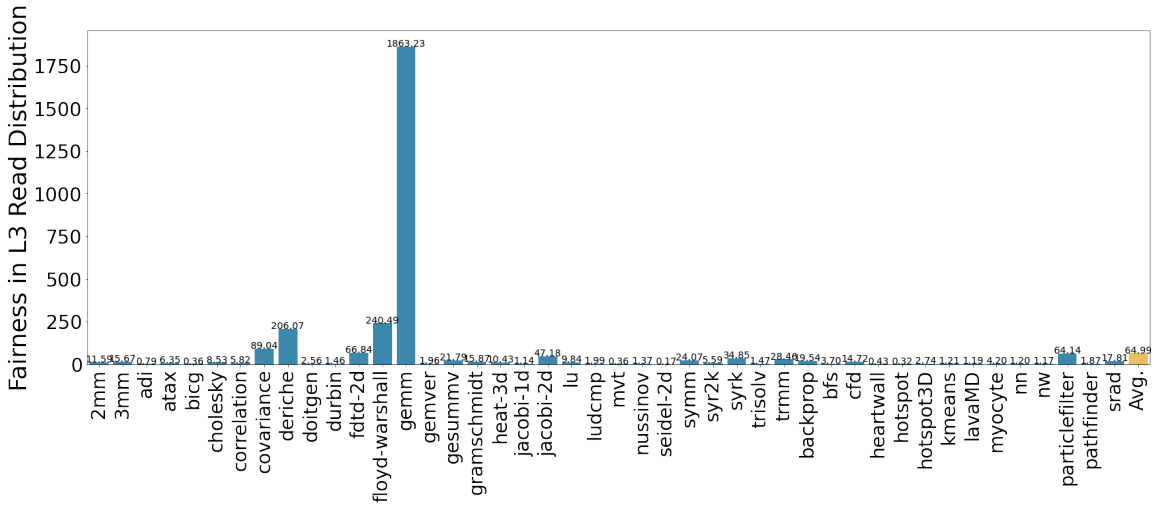


Figure 6.13: Fairness in load distribution among L3 caches for 108 processes job configuration

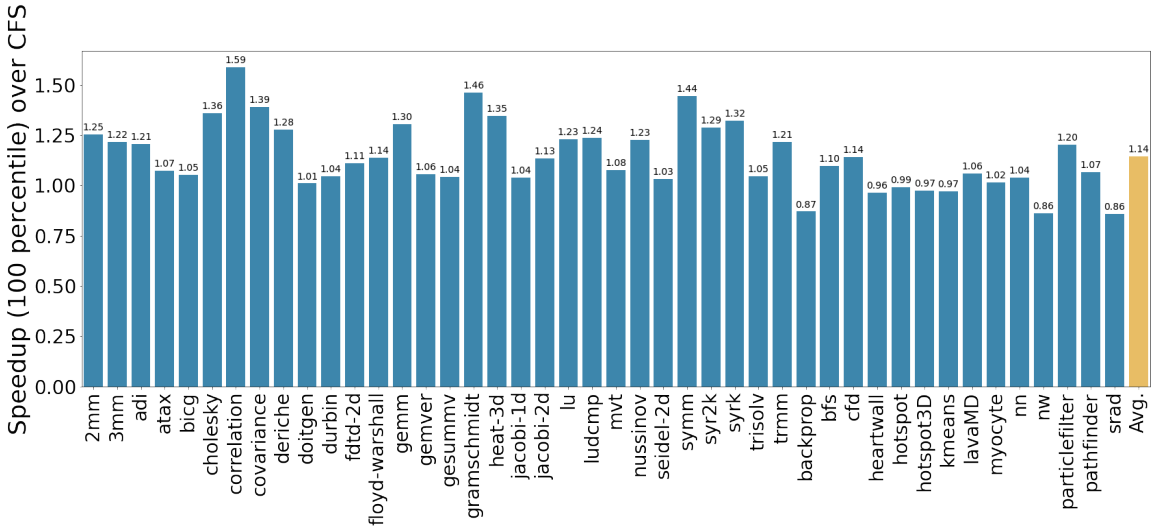


Figure 6.14: Speedup (100 th percentile) for 162 processes job configuration

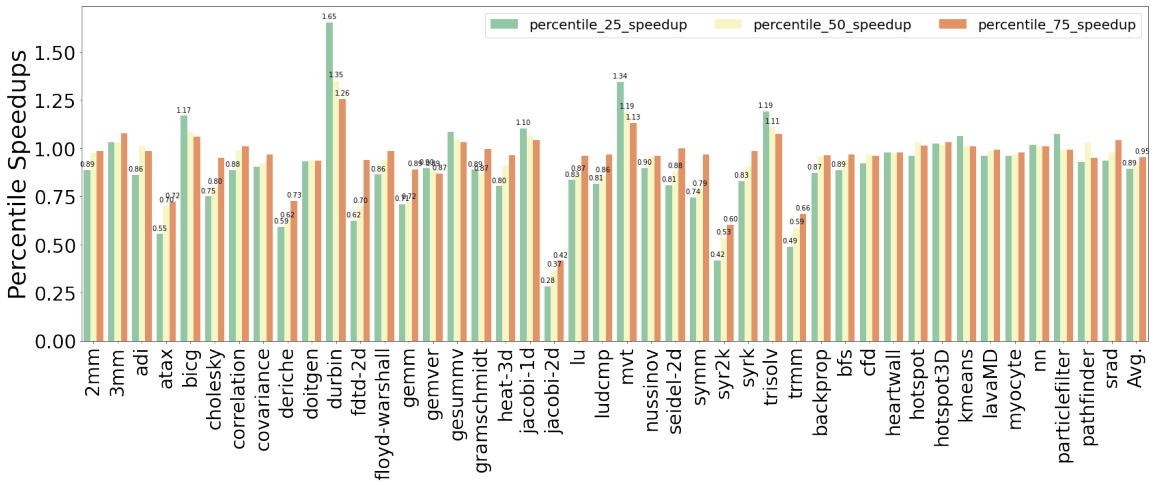


Figure 6.15: Speedup at 25, 50, 75 th percentile of processes completion for 162 processes job configuration

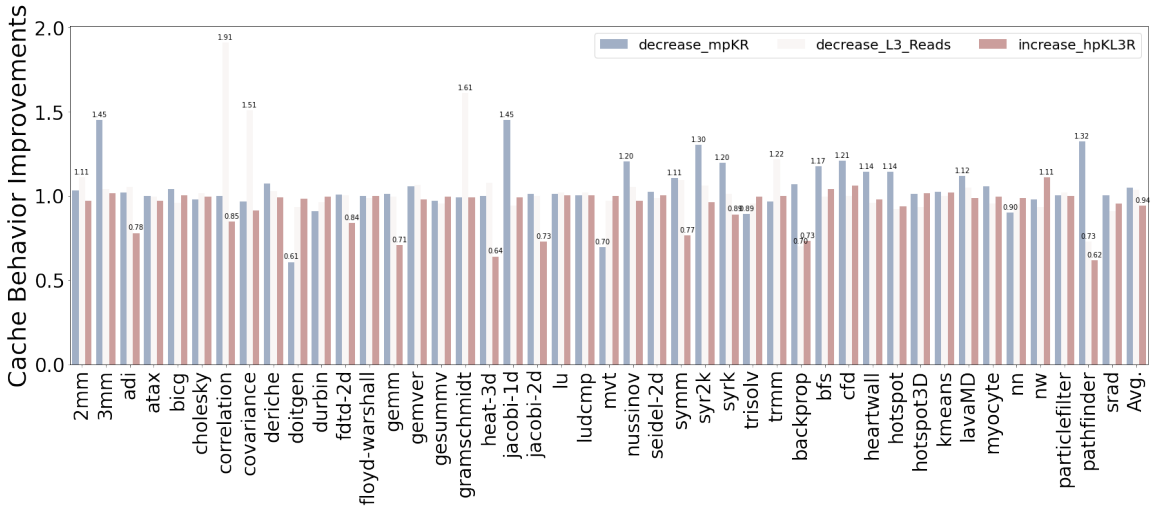


Figure 6.16: Cache Behavior in terms of cache misses, L3 reads and hits for 162 processes job configuration

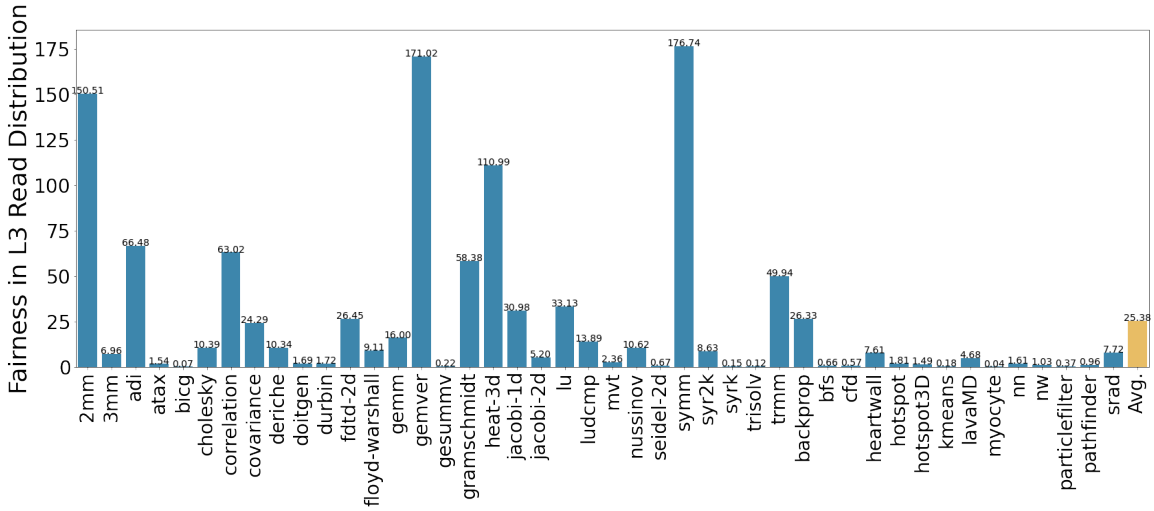


Figure 6.17: Fairness in load distribution among L3 caches for 162 processes job configuration

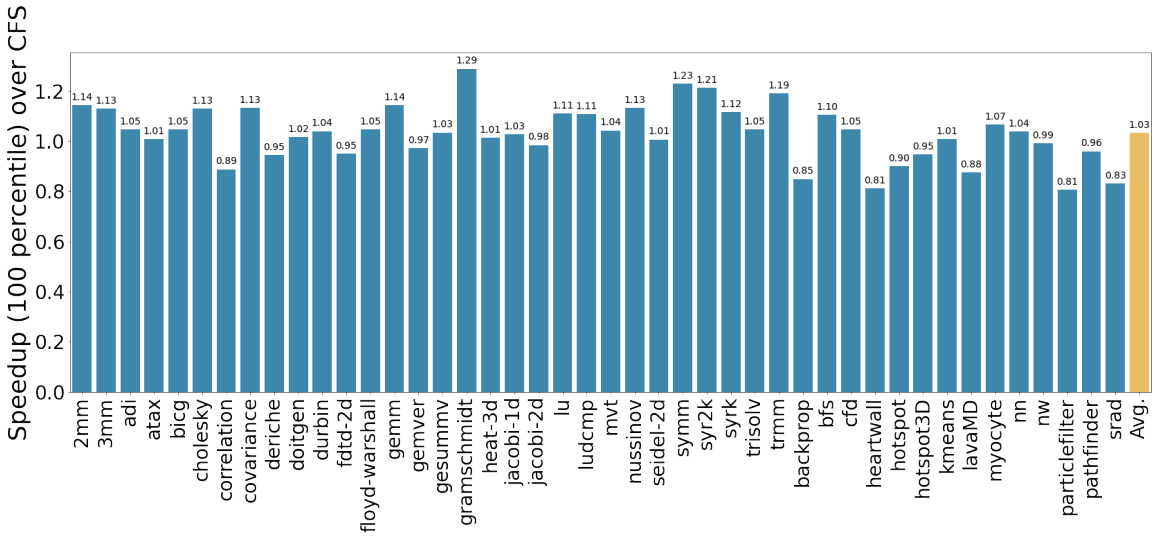


Figure 6.18: Speedup (100 th percentile) for 216 processes job configuration

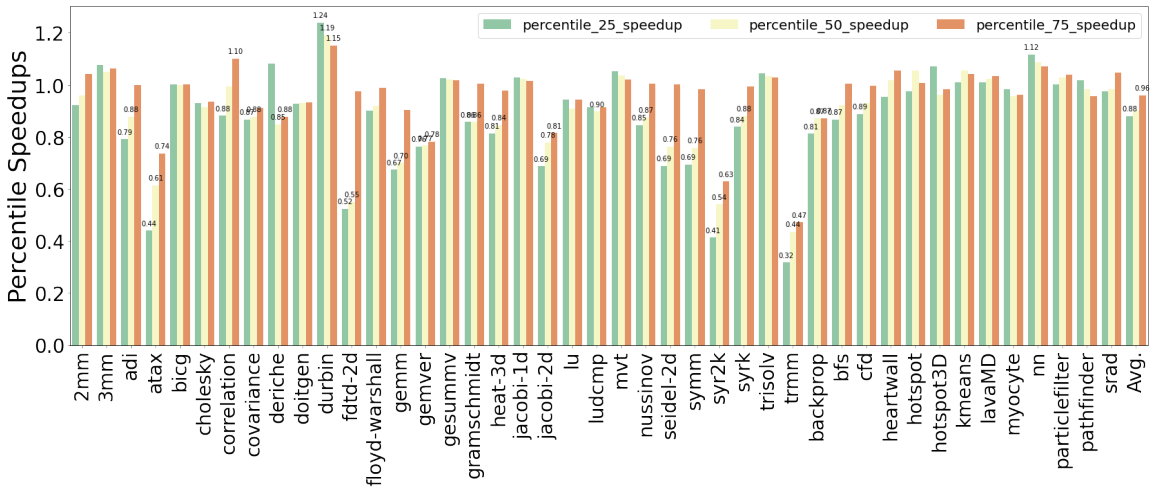


Figure 6.19: Speedup at 25, 50, 75 th percentile of processes completion for 216 processes job configuration

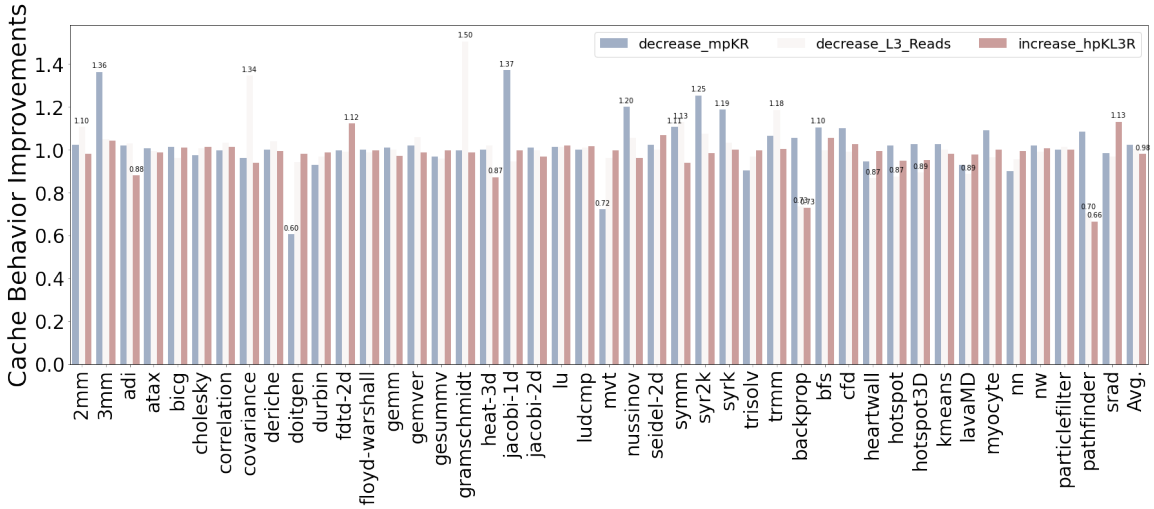


Figure 6.20: Cache Behavior in terms of cache misses, L3 reads and hits for 216 processes job configuration

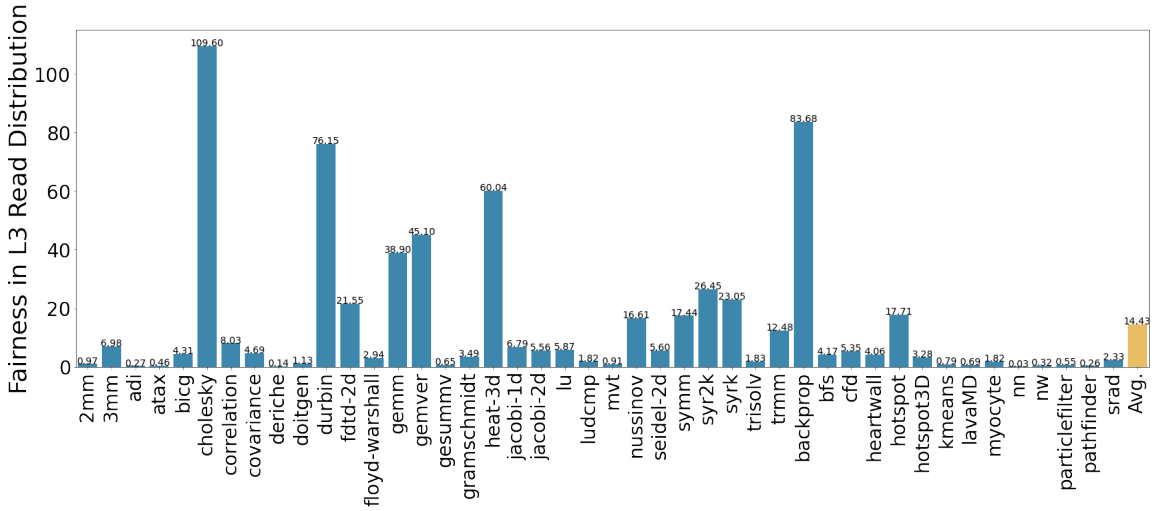


Figure 6.21: Fairness in load distribution among L3 caches for 216 processes job configuration

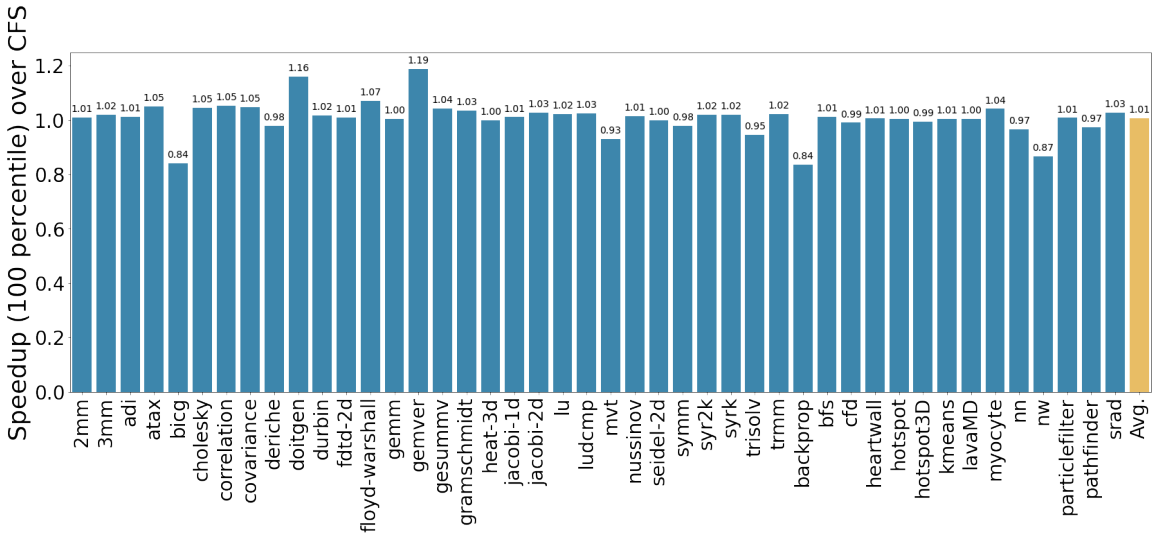


Figure 6.22: Speedup (100 th percentile) for 50 with xl processes job configuration

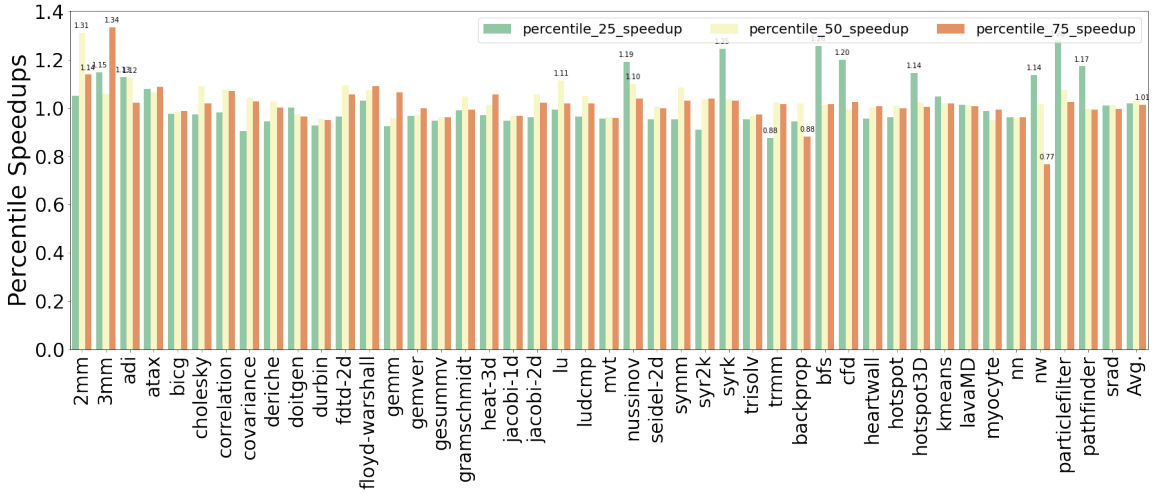


Figure 6.23: Speedup at 25, 50, 75 th percentile of process completion for 50 with xl processes job configuration

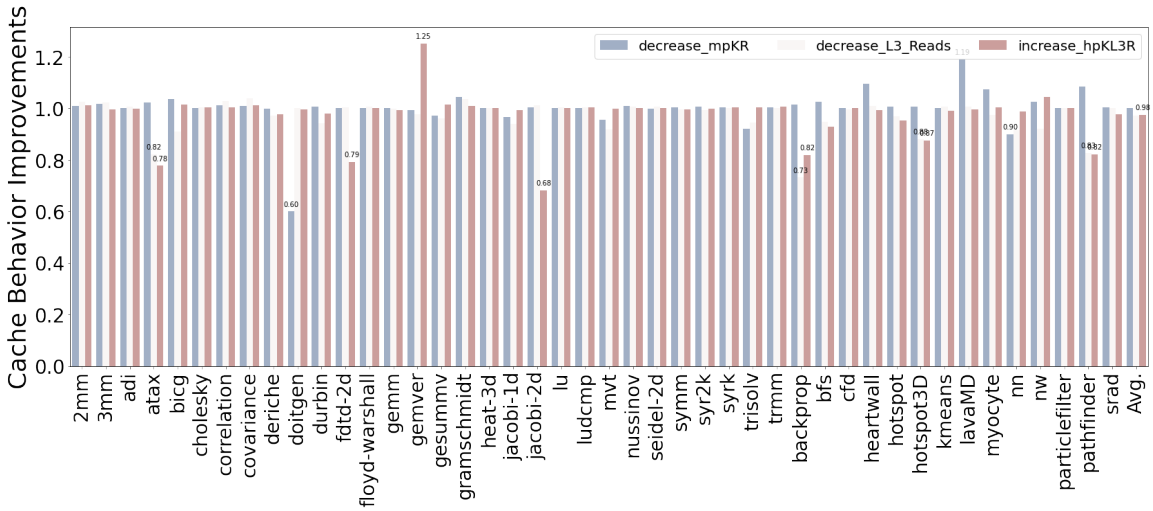


Figure 6.24: Cache Behavior in terms of cache misses, L3 reads and hits for 50 with xl processes job configuration

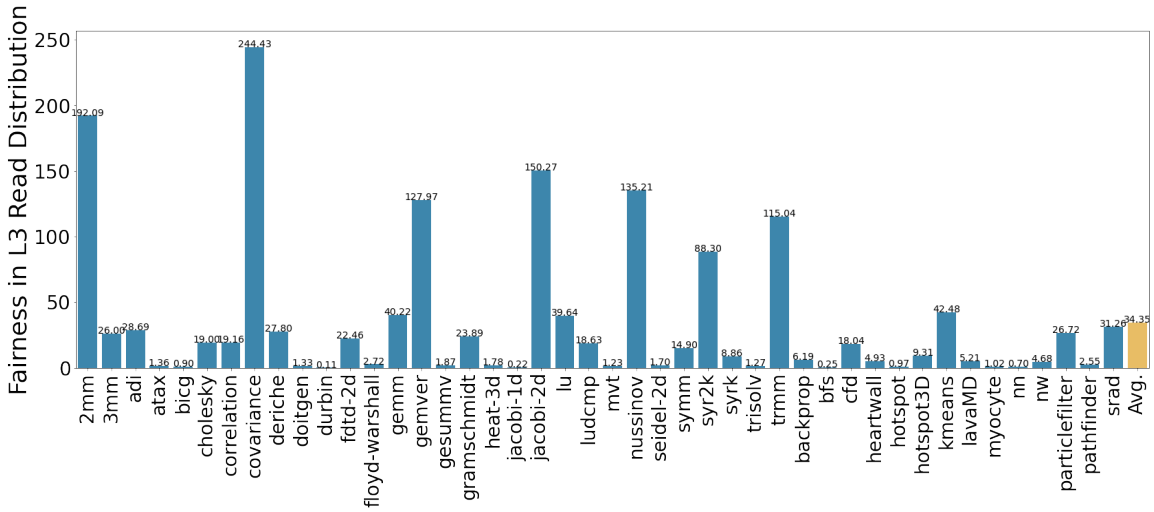


Figure 6.25: Fairness in load distribution among L3 caches for 50 with xl processes job configuration

and large input sizes excluding the extra-large input. For benchmarks in Rodinia, that did not have these many inputs, we used the corresponding inputs from BFS and Backprop in Rodinia as substitutes. Heartwall, Hotspot, and Hotspot3D benchmarks within Rodinia do not have a mini input, so we used Backprop with mini input as a substitute. All inputs for CFD in Rodinia are extra large inputs, memory footprint that usually exceed L3, hence CFD has BFS with large and small inputs and Backprop with medium and mini inputs in the six configurations. However, for the extra-large configuration mentioned below, the extra-large input benchmark in the configuration is CFD. The extra-large input is multiple orders of magnitude lengthy to complete than others, thus hiding any differences produced by intelligent scheduling decisions. The extra-large input processes must be fairly distributed among the sockets and CFS and Bellator both carry out similar co-location due to load balancing. We run another job configuration of 50 processes consisting of all mini, small, medium, large, and extra-large input sizes.

The limit on the minimum size of beacons to be sent to the scheduler is fixed at 0.001 seconds loop duration and 512 KB memory footprint. This results in more number of beacons and hence stresses the scheduler. The beacons can be reduced by increasing the limit of memory footprint to 32KB and duration to 0.01 secs, for example, which is equal to

the L1 size and enables the scheduler to focus on larger and fewer beacons as in the BES scheduler. We set the limit lower to stress Bellator to include the arbitration for the beacons that only occupy the shared L1 as well.

We report three different broad outcomes and behaviors for every configuration. All the readings are averaged over five runs. First, we report latency by reporting the speedup at the different percentile of process completion, i.e. at 25, 50,75, and 100^{th} percentile job completion. For example, in case of 27 job configuration, we report the speedup of 5^{th}, 13^{th}, 18^{th}, and 27^{th} process completion speedups. The speedup is calculated as the ratio of the execution time of the process configuration with CFS over Bellator. Secondly, we report the cache behavior in terms of the decrease in cache misses per 1000 references as **decrease_mpKR** (in ARM CPUs these cache misses refer to total L1 misses), decrease in total reads to L3 as **decrease_L3_Reads** (ThunderX2 has two sockets with two L3s which are measured as uncore events in perf. We sum the reads to both L3s and report the decrease in the number of reads to L3s in Bellator compared to CFS), increase in hits per 1000 cache-reads to L3 as **increase_hpKL3R**. In the cache behavior graph, for all the bar types the higher the bar is the better. However, note that the third bar **increase_hpKL3R** can be lower indicating incompetency to CFS, but in reality hits can be lower if the total reads itself is lower. Last, we report the distribution of load over the two L3 caches corresponding to the two sockets. Bellator always tries to best fit the processes' memory requirement in the available caches if possible and if not it also distributes the large processes that exceed L3 among the sockets thus ensuring L3 load distribution. We calculate the absolute difference in the number of cache-reads between the two L3s in the case of CFS and divide by the same in the case of Bellator. If Bellator has fairer distribution compared to CFS, then the fairness graph shows a huge ratio because the denominator is closer to 1 or 0 and vice-versa. Below we first describe the total speedup for the different configurations, followed by the percentile speedups, cache behavior improvements, and fairness in the L3 load distribution for all configurations.

The total speedup which is also how fast Bellator completed all the processes compared to CFS or in other words the latency of 100^{th} percentile of processes. The throughput of the scheduler is also a direct function of this number as 100^{th} percentile of the processes is completed in this time and the number of processes completed in unit time can be obtained by the ratio of the number of processes in the configuration by the execution time of the configuration. While the average speedup for Polybench and Rodinia together in 27 processes configuration is 3% negative as shown in Figure 6.2, that is a slowdown of 3%, in 54 processes it is 3% positive (Figure 6.6). For 108 and 162 processes configuration the average speedup for all the 45 benchmarks is 14% as shown in Figures 6.10 and 6.14, respectively. For 216 processes configuration, the speedup falls back to 3% as in Figure 6.18. At 108, 162 processes the total number of CPUs or hardware threads in use is 50% and 75% of the total hardware threads in the system, respectively. At these levels, the presence of empty and occupied hardware threads provide opportunities for intelligent co-location compared to other configurations in which either the empty CPUs are too few or too many. As noted earlier, in 50 processes with extra-large inputs, the scheduling is overshadowed by the long processes which are just fairly distributed among the sockets for most of its execution time by both CFS and Bellator compared to other processes that end relatively very early. However, we can still see Bellator performing better in many cases with improvements up to 19% and on average by 1%.

The percentile speedups at 25, 50, and 75^{th} percentile for 27 and 54 process configuration show Bellator closely tracing CFS as can be seen in Figures 6.3 and 6.7, respectively. However, at 108, 162, and 216 (shown in Figures 6.11, 6.15, and 6.19) the speedup at these points are much slower compared to CFS but the total speedup is reverse as seen earlier providing stronger evidence that CFS does benefit few processes over others and Bellator ensures the progress of all processes much strictly than CFS. In the case of 50 processes with extra-large inputs the percentile speedups include completion time of smaller processes which benefit from intelligent scheduling and we see gains over CFS as in Figure 6.23. The

imbalance in load distribution is convincingly evident in how unfair the L3 read distribution is in Figures 6.5, 6.9, 6.13, 6.17, 6.21, and 6.25, corresponding to all configurations. In all configurations, few benchmarks have L3 fairness values lower than 1 indicating that CFS is fairer in these cases in terms of L3 read load distribution. However, for the majority of the benchmarks, the values are greater than 1 and for few, the values are orders of magnitude above 1 indicating a huge difference in the perceived fairness between Bellator and CFS with Bellator being much fair on average.

In terms of cache behavior, 27 processes show a similar behavior to CFS with a slight increase in L3 reads (4%) with a decrease in L3 hits ($< 2\%$) and slightly increased cache misses ($<2\%$) as shown in Figure 6.4. For 54 processes the cache misses is reduced by 7% and L3 reads increase by $< 2.5\%$ and hits decrease by 3% as shown in Figure 6.8. The increased hits in L1 can make up for the loss in L3 performance. Similarly, in the case of 108 and 162 processes, cache misses are reduced by 4% and 4.8%, respectively, and L3 cache reads also reduce by 1.3% and 3.5%, respectively, but L3 cache hits per 1000 reads are reduced by 4.5% and 6%, respectively, as shown in Figures 6.12 and 6.16. For 216 processes, cache misses are reduced on average by 2% and L3 cache reads are same as CFS with 2% lower L3 hits as shown in Figure 6.20. In the case of 50 processes with extra large inputs, cache misses do not differ, but L3 reads increase by 3% and hits are reduced by 2.5% as shown in Figure 6.24. The cache behavior improvements answer drastic speedups or slowdowns and also most of the smaller improvements and degradation, however, they do not account for all changes mainly because ThunderX2 has a ring architecture for L3 cache and the access to the split L3 within each socket is non-uniform.

6.4 Related Work

Several works [50, 51, 52, 24, 2, 3] have addressed the problem of co-location through the use of performance counters. Among them, many works [50, 51, 24] mainly profile offline to determine co-run degradation and then use this information to decide on the processes

that must be co-located. Few works [52, 2] used this information along with online profiling to dynamically decide on co-location. These works are limited by the fixed set of off-line profiles that can be generated and a handicap to adapt to the dynamic environment. Few other works [3, 41] depend only on the on-line sampling of performance counters to drive the scheduling decisions of co-locating the processes and threads. However, we have seen these counter-based mechanisms fail in co-scheduling environments a simpler problem than co-location problem that involves three dimensions of what, when, and where compared to the two dimensions of what and when in co-scheduling.

Several works have predicted load via compiler for task-scheduling [46], reducing power consumption [47], and load-balancing [4] in large-clusters. Compilers have also been used to target energy-aware scheduling [6], instruction scheduling in embedded architectures, especially VLIW scheduling [53]. But none of these works use compilers towards predictive information passing to enable schedulers to find the right placement of processes such that the resources are managed very effectively to minimize latency like Bellator.

6.5 Conclusion

In this work, we proposed Bellator a beacon-based scheduler that pro-actively and dynamically co-locates processes on the system to minimize 100^{th} percentile latency while improving memory allocation fairness and maintaining computing fairness. The compiler inserted beacons predict the cache sensitivity and usage which are then used by Bellator to pro-actively place processes on the right cores to increase efficiency in overall resource usage. A prototype implementation of Bellator decreased overall 100^{th} percentile latency by 14% than CFS on Polybench and Rodinia together while running 108,162 processes concurrently on ThunderX2 servers.

CHAPTER 7
SECURE CO-EXECUTION USING BEACONS: THWARTING SIDE-CHANNEL ATTACKS

In this chapter, we provide a preview into how we can use beacons for security in a multi-tenant environment. We first describe and motivate the problem and then briefly describe a solution based on beacons and end with preliminary results.

7.1 Problem Motivation and Solution

Modern servers are multi-core machines that run multiple processes in parallel. These processes are isolated and protected from one another due to a separation of virtual address spaces which have different access permissions; in addition many servers also adopt virtual machines for multi-tenancy to achieve complete software-stack isolation. The purpose of isolation is to make the memory contents or private data of one process non-accessible to other processes. Despite such mechanisms, there have been attempts to gain access to private data. Private data has been leaked or attacked traditionally by exploiting memory corruption or deviating control flow of the processes through input strings [54, 55], for which strong defense mechanisms [56, 57, 58, 59, 60, 61, 62, 63] have been proposed. While these mechanisms safeguard against the faults in the program itself, side-channel attacks (that do not rely on return or jump oriented programming (ROP or JOP) [64, 65, 66, 67, 68, 69] are becoming ubiquitous. Side-channel attacks are a class of attacks that extract the secret key in cryptography algorithms by recording the changes (or differential behaviors) in the physical properties of the machine (which act as a side-channel). Attacks have used different physical properties of systems such as time [68, 67, 69], power consumption [70], memory consumption [71], sound [72] or electromagnetic emissions [73] to leak data.

Among various side-channel attacks that leverage different physical properties, time-

based attacks that utilize caches are most prevalent and are attract attackers for the following reasons:

- Caches are a major component of the data access pipeline and are used for reducing memory latency in all computer systems. Since caches are omnipresent, such attacks can be staged on a wide variety of systems.

- Cache-based side-channel attacks are easier to monitor, as external equipment is not required. The attacks can be carried out remotely without physically accessing the machine (which is a requirement in many other side-channel attacks such as electromagnetic emissions).

Currently, the literature describes three well-known cache-based side-channel attack techniques that retrieve the cryptography key: (i) Flush+Flush [69], (ii) Prime+Probe [67], and (iii) Flush+Reload [68]. These attacks force the victim's data out of the cache and then record the pattern in which the cache-sets or cache-lines were filled in by the victim. The adversary repeats this step and then analyzes the cache access pattern to obtain the victim's secret key, successfully demonstrated in [74, 75, 68, 76].

7.1.1 Cache-based Side-channel Attack Techniques

Several cache-based side-channel attacks (CSA) have been studied in the literature. Here, we focus on three well-known and recent cache-based side-channel attack techniques, also known as cache attack techniques (CATs), that recover the secret key from a cryptography algorithm.

1. **Flush+Reload:** This technique relies on identical cryptography code or data pages to be shared between the attacker and victim processes. The adversary also assumes a selected set of cache lines can be flushed through the invocation of certain instructions, e.g. the *clflush* instruction in X86. First, the attacker flushes a memory line from the cache. Then the attacker waits for a fixed interval during which the victim may access

the memory line, which brings the memory into the cache. After the wait, the attacker accesses the same memory line. If the victim accessed the memory line and cached it, the attacker's 2nd access duration will be much shorter compared to accessing the line from memory. By continuously repeating the above steps, the attacker records the pattern of memory accesses by the victim, which is later analyzed to deduce the secret key.

2. **Flush+Flush:** This technique is similar to the above Flush+Reload attack. It shares the same requirements and the same first step. The technique leverages the fact that the flush instruction (e.g. *clflush*) aborts early when the memory line is not in the cache. By exploiting this fact, the attacker, rather than accessing the memory line after the first flush (as in Flush+Reload), flushes the memory line again. If the victim did not access the memory line in the intervening period between the first and second flush, then the attacker's second flush will abort early. If the victim did access it, however, then the second flush evicts the memory line from all the caches, which takes more time. The attacker records all the cache lines accessed by the victim and then analyzes the differential behavior to crack the secret key as in Flush+Reload. The second flush not only checks if a memory line was accessed, but also sets up the cache to check if the line is accessed again, thus eliminating an extra step.

3. **Prime+Probe:** This technique does not have any prior setup requirements and hence can be much more pervasive. Before the attack is initiated, the attacker creates an eviction set, which is a set of known memory lines that will collide with a victim's cache set. In the prime step, the attacker fills the entire cache set with the memory lines from the corresponding eviction set. The attacker waits for a fixed interval of time, during which the victim may access a memory line from the cache set (thereby evicting a line from the eviction set). In the next probe step, the attacker accesses the eviction set again, checking if any memory line in the eviction set has been removed

from the cache by the victim. If the victim never evicted a cache line from the same cache set, then the accesses for each memory line in the eviction set will be short (and vice versa). The attacker records the cache access pattern to later analyze the secret key as in the above techniques. Similar to Flush+Flush, in Prime+Probe the probe step not only checks if any line in the eviction set was evicted, but also sets up the cache for the next probe by filling cache lines with the eviction set.

Regardless of which technique above is used, the attack causes a huge cache misses in the victim. This can serve as the basis for detecting the attack. The key questions to be answered however are: What is the expected behavior of cache-misses at a given program point during the application's dynamic execution (a no-attack scenario), and how can one carefully modulate the expected cache behaviors such that the departures from the same are successfully declared as attacks? Through a combination of compiler analyses that generate cache-miss models and by carefully controlling the scheduling decisions, this work successfully constructs such a solution. Before we peek into our scheme, we first provide a detailed survey of the existing solutions, citing their pros and cons.

7.1.2 Defense Mechanisms

The CATs shown so far directly target the secret key in the cryptography algorithms, and several works have tried to either prevent or to detect and mitigate these attacks.

Prevention Techniques

Several CAT prevention mechanisms focus on changing the cache designs, such as changing the cache replacement policy [77], encrypting the cache address [78, 79], or locking cache lines [80]. These solutions require changes to the hardware and hence do not apply to the already existing systems. In some cases, the solutions degrade the performance of the applications. Software-based hard isolation prevents the sharing of resources that contain sensitive data.

Cachebar [81] is a memory management subsystem that provides two main mechanisms against side-channel attacks. The first mechanism isolates pages by copying them, thus preventing sensitive cache lines from being shared among different processes. A process using shared library pages containing cryptography functions cannot be traced by an adversary using Flush+Reload or Flush+Flush CAT. The second mechanism limits the number of cache lines that a process can access, which inhibits the Prime+Probe CAT from exercising the entire eviction sets. The attacker is not able to retrieve all the accesses of the victim. However, these mechanisms are closely tied to the working of the above CATs. Creating private duplicate pages not only adds performance overheads but also sheds the benefits of shared libraries. Limiting the cache access adversely impacts the performance of genuine processes with overheads up to 25%.

StealthMem [82] allocates isolated pages called stealth pages to each process. These stealth pages map to unique cache sets such that no other page maps to these cache sets. Hence, another process cannot access the cache set that belongs to the Stealth pages. StealthMem assumes the confidential data and calculations are placed within these stealth pages. To adhere to this constraint, the source of the sensitive processes must be changed accordingly. The stealth pages create a partition of the cache. For four cores with a common last level cache (LLC), the shared cache size is reduced by 3%. The lost shared cache space increases with the number of cores that share the cache. The overheads reported on 4 cores with 6 VMs is at worse 11%. Modern machines comparatively have much higher core counts but do not have a correspondingly large a cache, which can significantly increase the overhead of StealthMem systems with 32 or 64 cores.

While Cachebar and StealthMem are hard isolation approaches, [83], a scheduler-based approach, is a soft isolation software solution that disrupts the recording of victims' cache access patterns by pre-empting other processes. The work analyzes the minimum run time guarantee and schedules the process with the corresponding time slices to avoid adversaries from recording the cache accesses. The scheduler also performs CPU state cleansing

between pre-emptions, in order to create soft isolation between processes. This technique, however, increases the latency of each process, and in server farm environments that over-provision cores [84], frequently de-scheduling the processes and idling the machines further decreases machine utilization and slows down execution. Thus, due to the reasons discussed above, both hard and soft isolation approaches are not practical.

Detection and Mitigation Techniques

There has been a lot of research on detection and mitigation. In particular, several researchers have studied the detection of these attacks. [85, 86] perform program analysis on binaries to model the secret key-dependent memory accesses and control-flow. The model is passed to an SMT solver to detect leakage areas that can be exploited by side-channel attacks. Program analysis and transformations ensure CPU cycles and cache misses/hits are independent of the secret data. For these leakage areas, however, program analysis and transformations disrupt the timing channels [87]. However, these transformations result in longer response and throughput times, with an average overhead of 50% and a worst case of 225%.

Several other techniques involve runtime mechanisms [88, 89, 90, 91] that use performance counters to check for anomalies in programs. Because of false positives in detecting anomalies, these runtime detection mechanisms do not mitigate the attack but leave it up to the system administrator for resolution. In addition, these techniques are closely tied to the cryptography algorithms for which they detect the attack. For example, SpyDetector [88] is a semi-supervised anomaly detection mechanism to detect side-channel attacks at runtime. The detection mechanism builds a clustering model that learns on cache misses, cache accesses, and the number of processes in the execution windows. The predicted workload level is passed to the clustering model, which raises an alarm for a possible attack if a window is not within the cluster. The clustering model is closely tied to the cryptography algorithms, thus weakening the detection efficacy for algorithms with modifications and for different workloads. Moreover, the mechanism must figure out the granularity of the

window that captures application phases, because different applications require different window sizes. This further reduces the generality of the mechanism.

In addition to the above, many software-based mechanisms that perform mitigation after detection have also been studied. CloudRadar [89] generates a cache-access profile of the cryptography applications with CATs. During program execution, it looks for behaviors that match the profile to flag an attack. If matched, CloudRadar migrates one of the processes or a known victim process to mitigate the attack. The execution profiles of the cryptography algorithms and the cache-hit and cache-miss profiles of CATs can be noisy, leading to many false anomalies. A new unknown attack with a slightly different profile or tricky modifications in the implementation of the known attacks decreases the strength of this mechanism. Also, the behavior of these applications can change in the presence of other co-executing applications, which can lead to a profile mismatch. Attacks may escape the detection radar, or the mechanism may flag false anomalies. Since co-executing applications, as well as variants of (known) attacks, are very likely in real execution environments, the defense mechanisms must handle them.

In summary, most of the above mechanisms are either hardware-based and do not apply to the existing machines or software-based solutions which reduce the efficiency of caches, increase latency, or closely tie themselves to the "environment" (i.e. the specific cryptography and cache attack algorithms, the applications' performance profiles, and some do not perform well in multi-tenant settings). Further, current hardware counter-based runtime detection mechanisms suffer from relatively high false positives and negatives. For example, SpyDetector has an F-score of 0.83 on Prime+Probe and Flush+Flush attacks. CloudRadar suffers from the burden of matching the execution windows of the victim and attacker, and finer window granularity raises false-positive rates up to 30%.

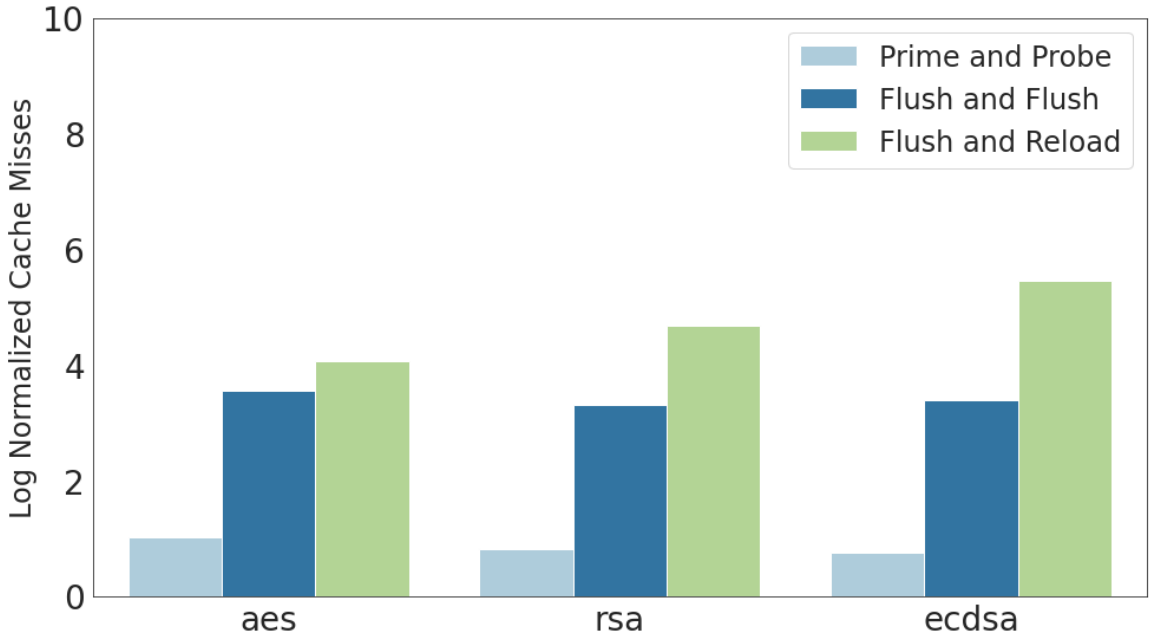


Figure 7.1: Log Normalized Cache Misses of Attacks

7.1.3 Beacon-based Solution

To overcome the above limitations, we propose a beacon-based solution which in a multi-tenant environment detects any cache-based side-channel attack on any program and then mitigates the attack by de-scheduling the plausible culprit, thereby avoiding any degradation of service (DS) attack. The potential culprit is scheduled back and is allowed to run to completion when all other processes have finished their execution to enable continuity. As noted earlier, our solution relies on the fact that the victims of CATs incur a significantly larger number of cache-misses compared to normal execution, as shown in Figure 7.1. The number of cache misses is at least five times that of the normal execution. To detect the cache behavior anomaly, we must first determine expected cache misses at certain program points during the application's normal (no attack) execution, and it must do this with significant accuracy. For this purpose, we must first generate a cache-miss model for every loop using compiler analysis. We use a linear model similar to the timing model described in chapter 2. The cache misses model replaces all other information in the beacons. During the execution, the beacon transmits the predicted values of cache misses of the loops to the scheduler.

The scheduler does not need extra memory footprint information because the *predicted cache-misses of a loop are equivalent to the cache footprint of the loop because the predicted cache-misses are mainly due to cold misses incurred by accessing unique memory references*. The scheduler then leverages these predicted cache-footprints to co-locate the processes such that the cache footprints of all the processes fit within the Last Level Cache (LLC) similar to Bellator. Since the cache footprints of scheduled processes fit in the cache, under normal execution (no attack), the expected cache miss behavior of each of the scheduled processes should remain unaltered (due to the lack of cache conflicts, that is, modern caches are highly associative and have very intelligent cache replacement policies, thus only capacity conflicts mostly occur in modern caches). In short, our solution, being the scheduler, can enforce (to a reasonable degree) non-collisions in multi-tenancy. Because of this, the departures from the predicted cache-misses must be attributable to some other reasons, viz. attacks. During the execution, the scheduler monitors the cache-misses to check if the cache miss prediction for a loop is violated for a given process. Upon encountering such a scenario, the scheduler through a careful search based on cache-misses counter isolates the culprit responsible for cache-misses in the victim.

We evaluated our solution on all of the above three CATs on OpenSSL's implementation of AES, RSA, and ECSDA cryptography algorithms in a multi-tenant environment. We were able to catch all the attacks on cryptography algorithms with no false positives.

7.2 Conclusion

In this chapter, we previewed how beacons can be used to thwart side-channel attacks. We explain the problem and shortcomings of the current techniques. We showed the side-effects of well-known side-channel attack techniques on cache misses of victim processes. We leverage this side-effect to develop a beacon-based scheduler that can effectively co-locate the processes such that the interference among them is minimized and then monitor for cache-misses to check for attacks based on beacon information. Preliminary results

showed that our beacon-based solution can detect and mitigate Prime+Probe, Flush+Reload, and Flush+Flush attacks on OpenSSL cryptography algorithms with no false positives or negatives. The full solution is outside the scope of this thesis.

CHAPTER 8
CONCLUSION

This thesis presents a compiler guided approach targeting different problems in scheduling. It starts with the problem of process migration in which the processes can be migrated while in the middle of a memory intense region causing the loss of warmed up caches. This thesis proposed a compiler-based approach called PinIt that determines the regions in the program to be pinned to a processor using the measure memory reuse density, a trade-off measure between the benefits from pinning for reuse of memory and scheduler flexibility for load balancing. The loops are pinned through a pin call to the library that decides on pinning and then calls Linux affinity API to pin the process to a processor. These pin calls are hoisted inter-procedurally. This technique used with non-preemption in an overloaded environment improves the performance of high-priority processes in mediabench workloads by 1.16x and 2.12x and in vision-based workloads by 1.35x and 1.23x on 8cores and 16cores, respectively, on average while completing the low-priority jobs in almost the same time as priority CFS thus demonstrating the optimization of multiple programs as an ensemble as against in isolation without undertaking infeasible inter-application analysis and without modifying the OS.

While PinIt communicated information about the program to the library to influence the scheduler, the information was specific to solving the problem of non-migration in critical regions and only influenced the underlying scheduler instead of the scheduler using the information directly. For bridging the gap between the compiler and the scheduler by providing the dynamic information about the executing processes to the scheduler to directly act upon, we developed Beacons by leveraging the library and the hoisting mechanism developed in PinIt. Beacons is a generic framework for placing and hoisting inter-procedurally the analyzed, profiled, or profiled and analyzed attributes that generate dynamic

information. Beacons also entail the library to communicate the generated information during runtime to the scheduler. First, the thesis showed how novel timing information is modeled and communicated through beacons along with the polyhedral-based memory footprint information and loop classification information based on static reuse distance defined in the thesis. The models are hoisted during compilation and are evaluated during runtime to generate the runtime attributes of the executing region. In this case, the duration of the loop, the memory footprint, and the type of the loop. The precision of the information also changes with the analyzability of the loops which is also sent to the scheduler to act appropriately. The thesis shows how the information is utilized by schedulers with different ambitions.

Next, we showed how beacons can be used to solve the problem of co-scheduling for maximizing the throughput of the system. The Beacon Enabled Scheduler (BES) in an environment with thousands of jobs and throughput of the system as the paramount goal can intelligently decide on the processes that must be dynamically co-scheduled together such that the resources such as cache is not over-subscribed leading to performance degradation. The BES scheduler based on the credibility of the beacon information collects performance counter information when required to back its decisions. It operates in either reuse or stream mode maximizing the total throughput of the system. A prototype implementation of the framework demonstrates improvements in throughput over CFS by up to 4.7x on ThunderX and up to 5.2x on ThunderX2 servers for consolidated workloads. The scheduler looks at the system as a whole with only the constraints on the total capacity of resources to decide on the co-schedule. In other words, the BES does not arbitrate the placement of the processes that are decided to be co-scheduled together. While maximizing the throughput of the system, the latency of each process can be sacrificed by the BES.

Second, to tackle the problems of co-location and latency that were left unsolved by the BES, we developed Bellator, which leverages the same information in beacons as the BES but uses the information to smartly co-locate the processes *without de-scheduling* them

such that all the resources are efficiently shared among the executing processes. Bellator knows the architecture of the system to decide on the placement of the processes. Bellator scheduling algorithm decides on the co-location based on the resource requirements, mainly checking for cache and memory bandwidth requirements. Bellator leverages the lessons learnt while developing PinIt and BES in deciding whether not to migrate and how to use the timing information. However, Bellator does not replace the two because they solve different problems. Bellator was used on co-locating different configuration of processes of the same benchmark with different input sizes and we observed that on ThunderX2 with 224 hardware threads we achieve lower 100^{th} percentile latency by 14% on average while executing 108 and 162 simultaneous processes and by 3% on average for 54 and 216 simultaneous processes while also drastically improving the fairness in load distribution on L3 caches.

Third, this thesis previewed how beacons can be used for security in a multi-tenant environment by successfully thwarting cache-based side-channel attacks. Our beacon-based solution was able to detect and mitigate all Prime+Probe, Flush+Reload, and Flush+Flush attacks on OpenSSL cryptography algorithms without any false positives. We replaced the timing, memory footprint, and loop classification beacon information used by the performance schedulers with just the predicted cache-misses information generated by a model similar to the timing model. Our scheduler used the cache-misses information to first smartly co-locate on the available sockets and then detected any cache-based side-channel attack and mitigated it by quarantining the adversary in all attempts.

To conclude, the thesis empirically demonstrates through the results obtained on a variety of benchmarks that a cross-stack approach under the guidance of the compiler can significantly improve the scheduling decisions leading to substantial performance improvements.

8.1 Usage of the system

We now discuss the usage of the developed systems in an integrated manner in Given the applications that are to be executed on a server farm environment.

In a server farm, we can first profile the applications for loop timings and cache-misses. The profile information can be used to develop the corresponding attribute models, which can then be embedded into the application using the beacons framework along with the statically analyzed memory footprint information and loop classification information. The statically analyzed memory footprint information is optional because it can be replaced with the cache-misses information which for modern machines with few conflict-misses within an application is equal to the memory requirement of the code region. The server farm usually consists of many machines (nodes) with each node of the configuration like that of the ThunderX2 machine and every node inter-connected with others through a high bandwidth inter-connection like the InfiniBand. At a higher level, the Beacon Enabled Scheduler can decide on the jobs that must be co-scheduled on the different nodes because BES does not concern with co-location within the system. By deciding on the processes to be co-scheduled on each node BES guarantees efficient dynamic resource usage of each node thus maximizing the throughput of the server farm. Within, each node we can run Bellator to efficiently co-locate the processes to minimize 100^{th} percentile latency and maximize fairness. This maximizes the throughput of each machine as well as the collective nodes. In an environment with high- and low-priority processes, PinIt can be enabled to limit the unnecessary migration of processes. The beacon information can be used to calculate the memory reuse density information by using the memory footprint and loop classification information for reuse and the timing information for instructions. If insufficient, we can always augment the beacons with the memory reuse density information. PinIt improves total machine utilization while maintaining priority. Within each node, Bellator can be enhanced with our security solution to detect and mitigate any cache-based side-channel

attacks. Thus, PinIt, BES, and Bellator with security together in an integrated manner can run a server farm securely while increasing machine utilization, maximizing throughput, and minimizing latency.

8.2 Future Work

The beacons framework opens up new horizons in terms of the problems that can be solved because of the systematic transfer of information from a compiler to the runtime manager, an entity in a different part of the software stack. In the thesis, we already previewed how the cache-based side-channel attacks could be thwarted from the beacon information thus entering the domain of security. Augmenting hardware performance counter information with the beacon information can enable new optimizations and reasoning like in our Beacon Enabled Scheduler (BES) and also our Beacon-based Secure scheduler. Beacons can also enable the use of new hardware features that expose control through software. For example, intel's cache allocation technology (CAT) exposes control of the cache to the software stack. Beacons, as is, can be used to take over the control of cache allocation for each process using intel CAT.

Beacons can also be applied towards multi-threaded applications in determining the resource requirements of threads in each process. For Open-mp programs that use **parallel pragma for**, the region is extracted into a separate function to which the thread id, which determines the loop bounds, is passed as a parameter. By using the thread id, each thread resource requirement can be sent to the scheduler. The scheduler, however, must be modified accordingly to do thread scheduling than just process scheduling. After multi-thread process scheduling on CPU, GPU scheduling of graphics kernels through beacons is also feasible with information from kernels. This can be broadened to enable beacons to benefit the processes from heterogeneous hardware on the machines. Heterogeneous hardware with different capabilities is picking up the pace and will soon become a norm in general-purpose machines. For such systems, the operating system scheduler will have to know what

kind of capabilities exist in each hardware type to arbitrate the processes' requirements, a problem especially hard if some kinds of hints in the program are unavailable. The reactive performance counter-based determination has a lot of shortcomings as listed earlier.

New kinds of cores with different computing capabilities and memory modules with different technologies such as DRAM, NVRAM are becoming mainstream. Beacon information that predicts the kind of computing within the code regions can help a scheduler like Bellator to co-locate processes on the cores with the right kind of computing power. NVRAM has persistent memory with different read and write speeds, unlike DRAM. By determining what arrays or what part of arrays are write-intensive versus read-intensive through program analysis can help in allocating the corresponding arrays or part of arrays in either DRAM or NVRAM, respectively. Other heterogeneous hardware includes hardware accelerators that solve specific problems very efficiently compared to the general cores on the machine. Although hardware accelerators require new programming models, the information such as the duration of the use of the hardware accelerator can help in the arbitration of the use of the hardware accelerator as a resource among many processes. Also if the accelerator includes shared resources within them, then these can also be arbitrated using the beacon information. Finally, exposing more scheduling information by the OS through APIs will enable deeper integration of Beacons towards the scheduling problems.

8.3 Other Works

In our other work BlankIt [92], we used compiler predicted call-chain for software debloating and security against non-control data attacks. Two other works model and harness non-determinism in parallel programs. [93] models non-determinism in pthread-based parallel programs by profiling architectural artifacts and uses the model to predict compiler optimization levels for reducing or increasing non-determinism. [94] models non-determinism in Software Transaction Memory (STM) by profiling commits and aborts and then uses the model to guide the STM in a less non-deterministic path to reduce timing variance.

REFERENCES

[1] C. Delimitrou, N. Bambos, and C. Kozyrakis, "Qos-aware admission control in heterogeneous datacenters," in *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, San Jose, CA: USENIX, 2013, pp. 291–296, ISBN: 978-1-931971-02-7.

[2] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13, Tel-Aviv, Israel: ACM, 2013, pp. 607–618, ISBN: 978-1-4503-2079-5.

[3] P. Tembey, A. Gavrilovska, and K. Schwan, "Merlin: Application- and Platform-aware Resource Allocation in Consolidated Server Systems," in *ACM Symposium on Cloud Computing (SOCC)*, Seattle, WA, Nov. 2014.

[4] V. Deodhar, H. Parikh, A. Gavrilovska, and S. Pande, "Compiler assisted load balancing on large clusters," in *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, ser. PACT '15, USA: IEEE Computer Society, 2015, 280–291, ISBN: 9781467395243.

[5] M. Novaes, V. Petrucci, A. Gamatié, and F. Quintão, *Compiler-assisted adaptive program scheduling in big.little systems*, 2019. arXiv: `1903.07038 [cs.PL]`.

[6] R. Xu, D. Zhu, C. Rusu, R. Melhem, and D. Mossé, "Energy-efficient policies for embedded clusters," in *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES '05, Chicago, Illinois, USA: Association for Computing Machinery, 2005, 1–10, ISBN: 1595930183.

[7] R. Kumar and D. M. Tullsen, "Compiling for instruction cache performance on a multithreaded architecture," in *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 35, Istanbul, Turkey: IEEE Computer Society Press, 2002, pp. 419–429, ISBN: 0-7695-1859-1.

[8] E. Bugnion, J. M. Anderson, T. C. Mowry, M. Rosenblum, and M. S. Lam, "Compiler-directed page coloring for multiprocessors," in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VII, Cambridge, Massachusetts, USA: ACM, 1996, pp. 244–255, ISBN: 0-89791-767-7.

[9] S. Carr, K. S. McKinley, and C.-W. Tseng, "Compiler optimizations for improving data locality," in *Proceedings of the Sixth International Conference on Architectural*

Support for Programming Languages and Operating Systems, ser. ASPLOS VI, San Jose, California, USA: ACM, 1994, pp. 252–262, ISBN: 0-89791-660-3.

[10] J. M. Anderson and M. S. Lam, "Global optimizations for parallelism and locality on scalable parallel machines," in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, ser. PLDI '93, Albuquerque, New Mexico, USA: ACM, 1993, pp. 112–125, ISBN: 0-89791-598-4.

[11] M. S. Squillante and E. D. Lazowska, "Using processor-cache affinity information in shared-memory multiprocessor scheduling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 2, pp. 131–143, Feb. 1993.

[12] N. Jammula, M. Qureshi, A. Gavrilovska, and J. Kim, "Balancing Context Switch Penalty and Response Time with Elastic Time Slicing," in *21st International Conference on High Performance Computing (HiPC)*, Goa, India, Dec. 2014.

[13] S. Chaudhry, P. Caprioli, S. Yip, and M. Tremblay, "High-performance throughput computing," *IEEE Micro*, vol. 25, no. 3, pp. 32–45, 2005.

[14] T. Kgil, S. D'Souza, A. Saidi, N. Binkert, R. Dreslinski, T. Mudge, S. Reinhardt, and K. Flautner, "Picoserver: Using 3d stacking technology to enable a compact energy efficient chip multiprocessor," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII, San Jose, California, USA: ACM, 2006, pp. 117–128, ISBN: 1-59593-451-0.

[15] V. Janapa Reddi, B. C. Lee, T. Chilimbi, and K. Vaid, "Web search using mobile cores: Quantifying and mitigating the price of efficiency," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10, Saint-Malo, France: ACM, 2010, pp. 314–325, ISBN: 978-1-4503-0053-7.

[16] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of OOPSLA'93*, A. Paepcke, Ed., ACM Press, 1993, pp. 91–108.

[17] S. Hofmeyr, J. Colmenares, C. Iancu, and J. Kubiatowicz, "Juggle: Proactive load balancing on multicore computers," Jan. 2011, pp. 3–14.

[18] T. C. K. Chou and J. A. Abraham, "Load balancing in distributed systems," *IEEE Transactions on Software Engineering*, vol. SE-8, no. 4, pp. 401–412, 1982.

[19] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 7, no. 2, pp. 279 –301, 1989.

[20] D. Grosu and A. T. C. and, "Load balancing in distributed systems: An approach using cooperative games," in *Proceedings 16th International Parallel and Distributed Processing Symposium*, 2002, 10 pp–.

[21] S. Peter, A. Schüpbach, P. Barham, A. Baumann, R. Isaacs, T. Harris, and T. Roscoe, "Design principles for end-to-end multicore schedulers," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Parallelism*, ser. HotPar'10, Berkeley, CA: USENIX Association, 2010, pp. 10–10.

[22] A. Mendelson and F. Gabbay, "The effect of seance communication on multiprocessing systems," *ACM Trans. Comput. Syst.*, vol. 19, no. 2, pp. 252–281, 2001.

[23] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, "Bottleneck identification and scheduling in multithreaded applications," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII, London, England, UK: ACM, 2012, pp. 223–234, ISBN: 978-1-4503-0759-8.

[24] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44, Porto Alegre, Brazil: ACM, 2011, pp. 248–259, ISBN: 978-1-4503-1053-6.

[25] T. Klug, M. Ott, J. Weidendorfer, and C. Trinitis, "Transactions on high-performance embedded architectures and compilers iii," in, P. Stenström, Ed., Berlin, Heidelberg: Springer-Verlag, 2011, ch. Autopin: Automated Optimization of Thread-to-core Pinning on Multicore Systems, pp. 219–235, ISBN: 978-3-642-19447-4.

[26] J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, and A. Fedorova, "The linux scheduler: A decade of wasted cores," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16, London, United Kingdom: Association for Computing Machinery, 2016, ISBN: 9781450342407.

[27] P. Radojkovic, V. Cakarevic, M. Moretó, J. Verdú, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero, "Optimal task assignment in multithreaded processors: A statistical approach," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, T. Harris and M. L. Scott, Eds., ACM, 2012, pp. 235–248.

[28] Z. Li, Y. Bai, H. Zhang, and Y. Ma, "Affinity-aware dynamic pinning scheduling for virtual machines," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, 2010, pp. 242–249.

[29] K. K. Pusukuri, R. Gupta, and L. N. Bhuyan, "Thread tranquilizer: Dynamically reducing performance variation," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, 46:1–46:21, Jan. 2012.

[30] A. Gujarati, F. Cerqueira, and B. Brandenburg, "Outstanding paper award: Schedulability analysis of the linux push and pull scheduler with arbitrary processor affinities," in *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, 2013, pp. 69–79.

[31] P. Jääskeläinen, P. Kellomäki, J. Takala, H. Kultala, and M. Lepistö, "Reducing context switch overhead with compiler-assisted threading," in *Proceedings of the 2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing - Volume 02*, ser. EUC '08, Washington, DC, USA: IEEE Computer Society, 2008, pp. 461–466, ISBN: 978-0-7695-3492-3.

[32] M. Lee and K. Schwan, "Region scheduling: Efficiently using the cache architectures via page-level affinity," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII, London, England, UK: ACM, 2012, pp. 451–462, ISBN: 978-1-4503-0759-8.

[33] J. Cong, P. Zhang, and Y. Zou, "Combined loop transformation and hierarchy allocation for data reuse optimization," in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '11, San Jose, California: IEEE Press, 2011, pp. 185–192, ISBN: 978-1-4577-1398-9.

[34] B. Bao and C. Ding, "Defensive loop tiling for shared cache," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, ser. CGO '13, Washington, DC, USA: IEEE Computer Society, 2013, pp. 1–11, ISBN: 978-1-4673-5524-7.

[35] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-based data reuse optimization for configurable computing," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '13, Monterey, California, USA: ACM, 2013, pp. 29–38, ISBN: 978-1-4503-1887-7.

[36] A. Jaleel, H. H. Najaf-abadi, S. Subramaniam, S. C. Steely, and J. Emer, "Cruise: Cache replacement and utility-aware scheduling," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII, London, England, UK: ACM, 2012, pp. 249–260, ISBN: 978-1-4503-0759-8.

[37] K. Beyls and E. H. D'Hollander, "Reuse distance-based cache hint selection," in *IN PROCEEDINGS OF THE 8TH INTERNATIONAL EURO-PAR CONFERENCE*, 2002, pp. 265–274.

[38] X. Shen, Y. Zhong, and C. Ding, "Locality phase prediction," in *ACM SIGOPS Operating Systems Review*, ACM, vol. 38, 2004, pp. 165–176.

[39] S. Padmanabha, A. Lukefahr, R. Das, and S. Mahlke, "Trace based phase prediction for tightly-coupled heterogeneous cores," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ACM, 2013, pp. 445–456.

[40] A. Fedorova and M. Seltzer, "Throughput-oriented scheduling on chip multithreading systems," Apr. 2019.

[41] A. Fedorova, M. I. Seltzer, C. Small, and D. Nussbaum, "Performance of multithreaded chip multiprocessors and implications for operating system design," in *Proceedings of the 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*, USENIX, 2005, pp. 395–398.

[42] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini, "Deepdive: Transparently identifying and managing performance interference in virtualized environments," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'13, San Jose, CA: USENIX Association, 2013, pp. 219–230.

[43] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto, "Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors," *ACM Computing Surveys*, vol. 45, no. 1, 2012.

[44] R Core Team, *R: A language and environment for statistical computing*, R Foundation for Statistical Computing, Vienna, Austria, 2014.

[45] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[46] L. Yang, J. M. Schopf, and I. Foster, "Conservative scheduling: Using predicted variance to improve scheduling decisions in dynamic environments," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, ACM, 2003, p. 31.

[47] S. Rele, S. Pande, S. Onder, and R. Gupta, "Optimizing static power dissipation by functional units in superscalar processors," in *Compiler Construction*, Springer, 2002, pp. 261–275.

[48] *Anandtech*, `https://www.anandtech.com/show/10353/investigating-cavium-thunderx-48-arm-cores/9`, Accessed: 2019 November 26, 2016.

[49] *Marvell*, https://www.marvell.com/documents/i8n9uq8n5zz0nwg7s8zz/marvell-thunderx2-energy-oil-and-gas-on-thunderx2-whitepaper, Accessed: 2019 November 26, 2019.

[50] Y. Jiang, X. Shen, J. Chen, and R. Tripathi, "Analysis and approximation of optimal co-scheduling on chip multiprocessors," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08, Toronto, Ontario, Canada: Association for Computing Machinery, 2008, 220–229, ISBN: 9781605582825.

[51] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," Mar. 2005, pp. 340–351, ISBN: 0-7695-2275-0.

[52] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV, Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 2010, 129–142, ISBN: 9781605588391.

[53] C. W. Kessler, "Compiling for vliw dsps," in *Handbook of Signal Processing Systems*, S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, Eds. Boston, MA: Springer US, 2010, pp. 603–638, ISBN: 978-1-4419-6345-1.

[54] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, 2007, pp. 552–561.

[55] T. K. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2011, Hong Kong, China, March 22-24, 2011*, 2011, pp. 30–40.

[56] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, Alexandria, VA, USA, November 7-11, 2005*, 2005, pp. 340–353.

[57] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, 2013, pp. 559–573.

[58] T. K. Bletsch, X. Jiang, and V. W. Freeh, "Mitigating code-reuse attacks with control-flow locking," in *Twenty-Seventh Annual Computer Security Applications Conference, ACSAC 2011, Orlando, FL, USA, 5-9 December 2011*, 2011, pp. 353–362.

[59] J. Criswell, N. Dautenhahn, and V. S. Adve, "Kcofi: Complete control-flow integrity for commodity operating system kernels," in *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, 2014, pp. 292–307.

[60] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque control-flow integrity," in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015.

[61] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, "Enforcing unique code target property for control-flow integrity," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, Toronto, Canada: ACM, 2018, pp. 1470–1486, ISBN: 978-1-4503-5693-0.

[62] X. Ge, W. Cui, and T. Jaeger, "GRIFFIN: guarding control flows using intel processor trace," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, 2017, pp. 585–598.

[63] Y. Liu, P. Shi, X. Wang, H. Chen, B. Zang, and H. Guan, "Transparent and efficient cfi enforcement with intel processor trace," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 529–540.

[64] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14, Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 361–372, ISBN: 978-1-4799-4394-4.

[65] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *arXiv preprint arXiv:1801.01203*, 2018.

[66] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD: USENIX Association, Aug. 2018, pp. 973–990, ISBN: 978-1-939133-04-5.

[67] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 605–622.

[68] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA: USENIX Association, Aug. 2014, pp. 719–732, ISBN: 978-1-931971-15-7.

[69] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+ flush: A fast and stealthy cache attack," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2016, pp. 279–299.

[70] C. Luo, Y. Fei, P. Luo, S. Mukherjee, and D. Kaeli, "Side-channel power analysis of a gpu aes implementation," in *2015 33rd IEEE International Conference on Computer Design (ICCD)*, 2015, pp. 281–288.

[71] S. Jana and V. Shmatikov, "Memento: Learning secrets from process footprints," in *2012 IEEE Symposium on Security and Privacy*, 2012, pp. 143–157.

[72] D. Genkin, A. Shamir, and E. Tromer, "Rsa key extraction via low-bandwidth acoustic cryptanalysis," in *Advances in Cryptology – CRYPTO 2014*, J. A. Garay and R. Gennaro, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 444–461, ISBN: 978-3-66.

[73] A. Sayakkara, N.-A. Le-Khac, and M. Scanlon, "A survey of electromagnetic side-channel attacks and discussion on their case-progressing potential for digital forensics," *Digital Investigation*, vol. 29, 43–54, 2019.

[74] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on aes, and countermeasures," *Journal of Cryptology*, vol. 23, no. 1, pp. 37–71, 2010.

[75] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of aes," in *Cryptographers' track at the RSA conference*, Springer, 2006, pp. 1–20.

[76] Y. Yarom and N. Benger, "Recovering openssl ecdsa nonces using the flush+ reload cache side-channel attack.," 2014.

[77] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel atacks," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 347–360, Jun. 2017.

[78] M. K. Qureshi, "Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2018, pp. 775–787.

[79] M. K. Qureshi, "New attacks and defense for encrypted-address cache," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19, Phoenix, Arizona: ACM, 2019, pp. 360–371, ISBN: 978-1-4503-6669-4.

[80] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 494–505, 2007.

[81] Z. Zhou, M. K. Reiter, and Y. Zhang, "A software approach to defeating side channels in last-level caches," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, Vienna, Austria: Association for Computing Machinery, 2016, 871–882, ISBN: 9781450341394.

[82] T. Kim, M. Peinado, and G. Mainar-Ruiz, "Stealthmem: System-level protection against cache-based side channel attacks in the cloud," in *Proceedings of the 21st USENIX Conference on Security Symposium*, ser. Security'12, Bellevue, WA: USENIX Association, 2012, p. 11.

[83] V. Varadarajan, T. Ristenpart, and M. Swift, "Scheduler-based defenses against cross-vm side-channels," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14, San Diego, CA: USENIX Association, 2014, 687–702, ISBN: 9781931971157.

[84] L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," *Computer*, vol. 40, no. 12, pp. 33–37, 2007.

[85] S. Wang, Y. Bao, X. Liu, P. Wang, D. Zhang, and D. Wu, "Identifying cache-based side channels through secret-augmented abstract interpretation," in *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA: USENIX Association, Aug. 2019, pp. 657–674, ISBN: 978-1-939133-06-9.

[86] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, "Cached: Identifying cache-based timing channels in production software," in *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC: USENIX Association, Aug. 2017, pp. 235–252, ISBN: 978-1-931971-40-9.

[87] M. Wu, S. Guo, P. Schaumont, and C. Wang, "Eliminating timing side-channel leaks using program repair," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ACM, 2018, pp. 15–26.

[88] Y. Kulah, B. Dincer, C. Yilmaz, and E. Savas, "Spydetector: An approach for detecting side-channel attacks at runtime," *International Journal of Information Security*, vol. 18, no. 4, pp. 393–422, 2019.

[89] T. Zhang, Y. Zhang, and R. B. Lee, "Cloudradar: A real-time side-channel attack detection system in clouds," in *International Symposium on Research in Attacks, Intrusions, and Defenses*, Springer, 2016, pp. 118–140.

[90] M. Sabbagh, Y. Fei, T. Wahl, and A. A. Ding, "Scadet: A side-channel attack detection tool for tracking prime-probe," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–8.

[91] M. Chiappetta, E. Savas, and C. Yilmaz, "Real time detection of cache-based side-channel attacks using hardware performance counters," *Appl. Soft Comput.*, vol. 49, no. C, 1162–1174, Dec. 2016.

[92] C. Porter, G. Mururu, P. Barua, and S. Pande, "Blankit library debloating: Getting what you want instead of cutting what you don't," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020, London, UK: Association for Computing Machinery, 2020, 164–180, ISBN: 9781450376136.

[93] G. Mururu, K. Ravichandran, A. Gavrilovska, and S. Pande, "Generating robust parallel programs via model driven prediction of compiler optimizations for non-determinism," in *49th International Conference on Parallel Processing - ICPP*, ser. ICPP '20, Edmonton, AB, Canada: Association for Computing Machinery, 2020, ISBN: 9781450388160.

[94] G. Mururu, A. Gavrilovska, and S. Pande, "Quantifying and reducing execution variance in stm via model driven commit optimization," in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO 2019, Washington, DC, USA: IEEE Press, 2019, 109–121, ISBN: 9781728114361.